



NGUYỄN HỮU ĐIỂN

THUẬT TOÁN VÀ LẬP TRÌNH

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

LỜI NÓI ĐẦU

Những năm trước khi lập trình VieTeX tôi toàn dùng C/C++ thu thập tài liệu nhiều nhưng không có thời gian để viết lại. Nay muốn viết lại thì sức khỏe không ổn định. Tôi đã cố gắng gom lại thành các tập lập trình theo chủ đề. Nội dung mỗi thuật toán bắt đầu từ lý thuyết đến lập trình bằng C/C++.

Cuốn sách viết ra không dành riêng cho các bạn học tin học, mà các bạ học toán, thầy cô giáo, các bạn thích tìm hiểu về thuật toán. Cũng như tôi bắt đầu có biết gì về lập trình đâu, tự học và chăm chỉ là thành công thôi. Tôi dùng trình biên dịch Dev-C++: <https://www.bloodshed.net/>

Hiện nay Dev-C++ cải tiến rất nhiều và chạy tốt với môi trường unicode. Những ví dụ trong tài liệu các bạn chép thẳng vào soạn thảo và biên dịch không cần cấu hình trình biên dịch.

Tôi đã làm các quyển sách:

1. Thuật toán và số học.
2. Thuật toán và dữ liệu.
3. Thuật toán xắp xếp
4. Thuật toán tìm kiếm
5. Thuật toán đồ thị,
6. Thuật toán quay lui
7. Thuật toán chia để trị
8. Thuật toán động
9. Thuật toán tham
10. Thuật toán nén
11. Một số đề thi Olympic Tin học.

Cuốn sách dành cho học sinh phổ thông yêu toán, học sinh khá giỏi môn toán, các thầy cô giáo, sinh viên đại học ngành toán, ngành tin học và những người yêu thích Toán - Tin. Trong biên soạn không thể tránh khỏi sai sót và nhầm lẫn mong bạn đọc cho ý kiến.

Hà Nội, ngày 25 tháng 2 năm 2022

Nguyễn Hữu Điển

NHỮNG KÝ HIỆU

Trong cuốn sách này ta dùng những kí hiệu với các ý nghĩa xác định trong bảng dưới đây:

\mathbb{N}	tập hợp số tự nhiên
\mathbb{N}^*	tập hợp số tự nhiên khác 0
\mathbb{Z}	tập hợp số nguyên
\mathbb{Q}	tập hợp số hữu tỉ
\mathbb{R}	tập hợp số thực
\mathbb{C}	tập hợp số phức
\equiv	dấu đồng dư
∞	dương vô cùng (tương đương với $+\infty$)
$-\infty$	âm vô cùng
\emptyset	tập hợp rỗng
C_m^k	tổ hợp chập k của m phần tử
:	phép chia hết
\nmid	không chia hết
$UCLN$	ước số chung lớn nhất
$BCNN$	bội số chung nhỏ nhất
deg	bậc của đa thức
IMO	International Mathematics Olympiad
APMO	Asian Pacific Mathematics Olympiad

NỘI DUNG

Lời nói đầu	iii
Những kí hiệu	iv
Mục lục	iv
Danh sách hình	viii
Danh sách bảng	ix
Chương 1. Số học và thuật toán	3
1.1. Khái niệm toán học và thuật toán	5
1.1.1. Tập hợp	5
1.1.2. Tập hợp số	8
1.1.3. Phép chia hết và chia có dư	11
1.1.4. Tính tổng và tích	12
1.1.5. Tính lũy thừa, logarit và căn	16
1.1.6. Bài tập	17
1.1.7. Giai thừa và hồi qui	17
1.1.8. Ma trận	19
1.1.9. Tìm số chữ số của một tích	22
1.2. Số nguyên tố	23
1.2.1. Kiểm tra một số có phải là số nguyên tố	26
1.2.2. Sàng Eratosten. Tìm số nguyên tố trong khoảng	28
1.2.3. Phân tích một số thành tích thừa số nguyên tố	33
1.2.4. Tìm số lượng số không trong kết quả phép nhân	34
1.3. Số mensen và số hoàn thiện	36
1.3.1. Số mensen	36
1.3.2. Số hoàn thiện	38
1.4. Những hệ số đa thức, tam giác Pascal và giai thừa	42
1.5. Hệ số đếm và sự biến đổi hệ	47
1.5.1. Chuyển hệ cơ số 10 sang cơ số p	50
1.5.2. Chuyển hệ cơ số p vào cơ số 10. Sơ đồ Horner	53

1.6. Chữ số la mã	55
1.6.1. Biểu diễn số thập phân thành chữ số La mã	56
1.6.2. Chuyển đổi chữ số La Mã sang số thập phân	58
1.7. Hồi quy và lặp lại	59
1.7.1. Tính giai thừa	60
1.7.2. Dãy Phibonacci	62
1.7.3. Ước số chung lớn nhất và thuật toán Euclid	69
1.7.4. Bội số chung nhỏ nhất	72
1.7.5. Trả lại giá trị từ đệ quy và dùng biến	74
1.8. Thuật toán đếm cơ sở	78
1.8.1. Hoán vị	79
1.8.2. Mã hóa và giải mã	84
1.8.3. Hoán vị lặp lại	87
1.9. Chính hợp	88
1.9.1. Các dạng chính hợp và cách sinh ra	88
1.10. Tổng bằng không	92
1.11. Tổ hợp	94
1.12. Biểu diễn số thành tổng	97
1.12.1. Tạo ngắt số dưới dạng tổng của các số đã cho	97
1.12.2. Sinh ra tất cả biểu diễn một số như là tích của các số tự nhiên	99
1.12.3. Sinh ra tất cả biểu diễn một số như là tổng của các số tự nhiên	101
1.12.4. Phân hoạch một tập hợp	103
1.13. Đánh giá và độ phức tạp của thuật toán	105
1.13.1. Lượng dữ liệu đầu vào	108
1.13.2. Ký hiệu tiệm cận	109
1.13.3. Tính chất và ví dụ của $O(F)$	111
1.13.4. Tính chất và ví dụ về Θ	113

1.13.5. Hàm tiệm cận và số thực	116
1.13.6. Xác định độ phức tạp của một thuật toán	117
1.14. Phương trình đặc trưng	127
1.14.1. Phương trình thuần nhất tuyến tính với nghiệm đơn giản	
127	
1.14.2. Phương trình thuần nhất tuyến tính với nhiều nghiệm .	
129	
1.14.3. Phương trình tuyến tính không thuần nhất	131
1.15. Các kỹ thuật đặc biệt để phân tích thuật toán	135
1.15.1. Sử dụng phong vũ biểu	135
1.15.2. Phân tích khâu hao	136
1.15.3. Định lý cơ bản	137
1.15.4. Các vấn đề về ký hiệu tiệm cận	139
1.16. Các câu hỏi và bài tập	140
1.16.1. Các bài toán về số, chuỗi, hàm	140
1.16.2. Bài toán ma trận và bài toán chung	146
1.16.3. Bài toán tổ hợp	148

Chương 2. Cấu trúc dữ liệu và thuật toán	153
2.1. Giới thiệu	154
2.2. Danh sách, ngăn xếp và hàng đợi	157
2.2.1. Danh sách	157
2.2.2. Ngăn xếp	159
2.2.3. Hàng đợi	160
2.2.4. Hàng đợi hai đầu	161
2.3. Thực hiện cụ thể cấu trúc trên	161
2.3.1. Danh sách	161
2.3.2. Ngăn xếp	162
2.3.3. Hàng đợi	164
2.4. Thể hiện cấu trúc động	168
2.4.1. Đưa vào một phần tử	169
2.4.2. Cuộn qua một danh sách	169
2.4.3. Đưa vào sau một phần tử được chỉ định bởi một chỉ dẫn đã cho	171
2.4.4. Đưa vào phía trước một phần tử được chỉ định bởi một thư mục	171
2.4.5. Xóa theo khóa mặc định và con trỏ lên đầu danh sách ...	172
2.4.6. Xóa một phần tử được chỉ định bởi một thư mục ..	173
2.5. Cây nhị phân	178
2.5.1. Tìm kiếm bằng chìa khóa	183
2.5.2. Thêm vào một đỉnh mới	184
2.5.3. Xóa một đỉnh bằng từ khóa đã cho	184

2.5.4. Thu thập thông tin	189
2.5.5. Bài tập	191
2.6. Cây cân đối	192
2.6.1. Vòng xoay. Cây đỏ và đen	196
2.6.2. B-Cây	198
2.7. Bảng băm (H-bảng)	201
2.7.1. Hàm băm (H-hàm số)	202
2.7.2. Sự xung đột	203
2.7.3. Hàm băm cổ điển	205
2.7.4. Đối phó với xung đột	214
2.7.5. Triển khai bảng băm	218
2.8. Câu hỏi và bài tập	226

2.26	Cây Fibonacci của hàng 3.	194
2.27	Cây Fibonacci của hàng 4.	194
2.28	Cây Fibonacci của hàng 5.	195
2.29	Xoay phải (A), trái (B), xoay trái-phải (C) và xoay phải-trái (D) quay.	197
2.30	Ví dụ về cây đỏ-đen.	198
2.31	2-3-4 cây.	199
2.32	B-cây hàng 5.	200
2.33	Hàm băm khớp với địa chỉ của từng khóa trong bảng băm.	203
2.34	χ^2 so sánh các hàm băm trên PIN, $n = 1031, m = 1031000$.	207
2.35	Bao gồm tuần tự của một phần tử. Cho phép xung đột với thử nghiệm tuyến tính với bước 1.	215
2.36	Giải quyết xung đột bằng cách phân bổ bộ nhớ bổ sung.	216
2.37	Kết nối động: bảng băm, sau khi bao gồm các phần tử 234, 235, 567, 123, 534, 647.	217

Chương 3. Sắp xếp dữ liệu và thuật toán	231
3.1. Giới thiệu	232
3.2. Sắp xếp theo so sánh	234
3.2.1. Cây so sánh	234
3.2.2. Sắp xếp theo cách chèn	236
3.2.3. Sắp xếp theo thứ tự chèn với bước giảm dần. Thuật toán của Shell	240
3.2.4. Phương pháp bong bóng	243
3.2.5. Sắp xếp bằng cách lắc	245
3.2.6. Nhanh chóng phân loại Hoar	246
3.2.7. Phương pháp Thỏ và Rùa	256
3.2.8. Sắp xếp theo lựa chọn trực tiếp	259
3.2.9. Sắp xếp kim tự tháp Williams	260
3.2.10. Độ phức tạp về thời gian tối thiểu của việc sắp xếp theo cách so sánh	266
3.2.11. Bài tập	268
3.3. Sắp xếp theo sự biến đổi	268
3.3.1. Sắp xếp theo tập hợp	268
3.3.2. Sắp xếp theo số đếm	272
3.3.3. Sắp xếp theo bit	276
3.3.4. Phương pháp hệ đếm số	282
3.3.5. Sắp xếp theo hoán vị	286
3.4. Sắp xếp song song	288
3.4.1. Nguyên tắc về số không và số một	292

3.4.2. Trình tự bitonic	294
3.4.3. "Rõ ràng một nửa"	295
3.4.4. Sắp xếp chuỗi bitonic.....	295
3.4.5. Sắp xếp sơ đồ hợp nhất.....	296
3.4.6. Sắp xếp sơ đồ phân loại	296
3.4.7. Sơ đồ phân loại chuyển vị	297
3.4.8. Sơ đồ sáp nhập Batcher chẵn lẻ.....	298
3.4.9. Lược đồ sắp xếp chẵn-lẻ.....	298
3.4.10. Lược đồ hoán vị	299
3.5. Câu hỏi và bài tập	301

Chương 4. Tìm kiếm và thuật toán	303
4.1. Giới thiệu	303
4.2. Tìm kiếm liên tiếp	306
4.2.1. Tìm kiếm liên tiếp trong danh sách đã sắp xếp	309
4.2.2. Tìm kiếm liên tục với sự sắp xếp lại	311
4.3. Tìm kiếm theo từng bước. Tìm kiếm bậc hai	313
4.4. Tìm kiếm nhị phân	317
4.5. Tìm kiếm Fibonacci	323
4.6. Tìm kiếm nội suy	326
4.7. Câu hỏi và bài tập	328

Chương 5. Lý thuyết đồ thị và thuật toán	330
5.1. Các khái niệm cơ bản	331
5.2. Trình bày và các thao tác đơn giản với đồ thị.....	338
5.2.1. Danh sách các cạnh	338
5.2.2. Ma trận lân cận, ma trận trọng số.....	339
5.2.3. Danh sách những cạnh kề	340
5.2.4. Ma trận tỷ lệ giữa các đỉnh và cạnh	341
5.2.5. Các thành phần kết nối	341
5.2.6. Xây dựng và các hoạt động đơn giản với đồ thị	342
5.3. Trình duyệt trên đồ thị	344
5.3.1. Duyệt theo chiều rộng.....	345
5.3.2. Duyệt theo chiều sâu	350
5.4. Đường dẫn, chu trình và dòng chảy tối ưu trong đồ thị	353
5.4.1. Các ứng dụng trực tiếp của thuật toán duyệt	354
5.4.2. Đường tối ưu trong đồ thị.....	363
- Bất đẳng thức của tam giác	365
- Thuật toán Ford-Bellman	366
- Thuật toán của Floyd	367
- Thuật toán tổng quát của Floyd	371
- Thuật toán Dijkstra	374
- Lũy thừa ma trận của phần tử lân cận	379
- Thuật toán Warshal và ma trận khả năng tiếp cận	380
- đường đi dài nhất trong thị chu trình	381

- Đường đơn dài nhất giữa hai đỉnh trong bất kỳ đồ thị nào ...	
386	
5.4.3. Chu trình	386
- tìm kiếm một tập hợp các chu trình cơ bản	386
- chu trình tối thiểu qua đỉnh.....	390
5.4.4. Các chu trình Hamilton. Bài toán đường thương mại	
391	
5.4.5. Chu trình Euler	395
5.4.6. Luồng trong đồ thị	400
- Lưu lượng luồng cực đại	402
- Nhiều nguồn và người tiêu dùng	408
- Công suất của các đỉnh	408

Chương 5. Lý thuyết đồ thị và thuật toán	410
5.5. Tính bắc cầu và cách xây dựng. Sắp xếp cầu trúc liên kết	
411	
5.5.1. Đóng bắc cầu. Thuật toán Worschal	411
5.5.2. Định hướng bắc cầu	413
5.5.3. Giảm bắc cầu	418
5.5.4. Kiểm soát công ty	420
5.5.5. Bài tập	422
5.5.6. Sắp xếp theo cấu trúc liên kết	422
5.5.7. Sắp xếp tô pô đầy đủ	426
5.5.8. Bổ sung một đồ thị xoay chiều thành một đồ thị được liên kết yếu	430
5.5.9. Xây dựng đồ thị của các đỉnh đã cho	431
5.6. Khả năng tiếp cận và liên thông	432
5.6.1. Các thành phần liên thông	433
5.6.2. Các thành phần liên kết mạnh trong đồ thị định hướng ..	
435	
5.6.3. Điểm phân chia trong một đồ thị không định hướng	
439	
5.6.4. k—liên thông của đồ thị không định hướng	443
5.7. Các tập hợp con tối ưu và các tâm của đồ thị	444
5.7.1. Cây bao phủ tối thiểu	444
- Thuật toán Kruskal	445
- Thuật toán của Prim	451
- Cây bao phủ tối thiểu một phần	454

5.7.2. Tập đính độc lập	455
- Tập hợp độc lập tối đa	456
5.7.3. Tập đính trội	459
5.7.4. Tập cơ sở	463
5.7.5. Tâm, bán kính và đường kính	466
- p -tâm và p -bán kính	470
5.7.6. Kết hợp cặp. Kết hợp cặp tối đa	475
5.8. Tô màu và đồ thị phẳng	477
5.8.1. Tô màu đồ thị và sắc số	477
- Giới hạn dưới cùng của số sắc	478
- Tìm sắc số đính	478
5.8.2. Đồ thị phẳng	479
5.9. Câu hỏi và bài tập	481

Chương 6. Thuật toán quay lui	488
6.1. Phân loại bài toán	489
6.1.1. Phức tạp về thời gian	489
6.1.2. Độ phức tạp tính toán theo bộ nhớ	490
6.1.3. Bài toán không thể giải được	490
6.1.4. Các ví dụ	490
6.2. Bài toán NP-đầy đủ	495
6.3. Tìm kiếm với quay lui	498
6.3.1. Sự thỏa mãn của một hàm Boolean	500
6.3.2. Tô màu đồ thị	507
6.3.3. Đường đi đơn dài nhất trong đồ thị chu trình	511
6.3.4. Đường đi quân ngựa	514
6.3.5. Bài toán tám quân Hậu	519
6.3.6. Thời khóa biểu của trường học	524
6.3.7. Dịch mật mã	529
6.4. Phương pháp nhánh và ranh giới	534
6.4.1. Bài toán ba lô (lựa chọn tối ưu)	535
6.5. Các chiến lược tối ưu cho trò chơi	539
6.5.1. Trò chơi "X" và "O"	541
6.5.2. Nguyên tắc minimum và maximum	546
6.5.3. Nhát cắt alpha-beta	548
6.5.4. Duyệt alpha-beta đến một độ sâu nhất định	551
6.6. Câu hỏi và bài tập	553

Chương 7. Thuật toán chia để trị	560
7.1. Giới thiệu	560
7.2. Tìm phần tử lớn nhất thứ k	561
7.3. Phần tử trội	572
7.4. Hợp nhất các mảng đã sắp xếp	590
7.5. Sắp xếp theo hợp nhất	597
7.6. Nâng nhanh lũy thừa	607
7.7. Thuật toán Strassen để nhân nhanh các ma trận	610
7.8. Nhân nhanh các số dài	616
7.9. Bài toán tháp Hà Nội	620
7.10. Tổ chức giải vô địch bóng đá	624
7.11. Sự dịch chuyển tuần hoàn của các phần tử mảng	631
7.12. Câu hỏi và bài tập	636

1.1. Khái niệm toán học và thuật toán

Tài liệu trong đoạn này nhằm mục đích giúp người đọc làm quen với các ký hiệu, thuật ngữ, khái niệm và các tính chất cơ bản của một số đối tượng toán học được sử dụng trong cuốn sách. Khuyến nghị của chúng ta, ngay cả đối với những người đã có nhiều kinh nghiệm với các thuật toán máy tính, là hãy đọc kỹ chương giới thiệu này để những điều chưa giải thích được từ nó càng ít càng tốt. Đây là điều kiện cần để có thể hiểu đầy đủ hơn về tài liệu. Tất nhiên, không nhất thiết phải nhớ tất cả các thuật ngữ và định nghĩa. Chỉ cho thấy một chút “hiểu biết nhỏ” về ngôn ngữ chặt chẽ và trừu tượng của toán học.

1.1.1. Tập hợp

Khái niệm tập hợp là chính và không được định nghĩa chặt chẽ bởi các khái niệm toán học khác. Nó được nhận thức một cách trực quan và thường được làm rõ qua các ví dụ.

Giả thiết rằng một tập hợp được đưa ra khi một hệ thống (tập hợp) các đối tượng được đưa ra, hầu hết thường được thống nhất trên cơ sở một số đặc điểm, thuộc tính chung hoặc một số quy tắc nào đó. Ví dụ, những con ong tạo thành một tập hợp trong một tổ ong, các đỉnh của một đa giác, các đường thẳng đi qua một điểm cho trước, nghiệm nguyên của phương trình $ax^2 + bx + c = 0$, v.v.

Các chữ cái viết hoa của bảng chữ cái Latinh thường được sử dụng làm ký hiệu cho một tập hợp: A, B, C, \dots . Các đối tượng tham gia vào một tập hợp được gọi là các phần tử của tập hợp và thường được đánh dấu bằng các chữ cái Latinh viết thường (hoặc bằng cách đánh chỉ mục thích hợp). Chúng ta sẽ tránh định nghĩa chặt chẽ về tập chỉ mục là gì, nhưng thường đối với các phần tử của tập A , chúng ta sẽ sử dụng chuỗi được lập chỉ mục a_1, a_2, \dots, a_n và chúng ta sẽ biểu thị nó theo cách này: $A = \{a_1, a_2, \dots, a_n\}$

Thực tế là phần tử $a_i, i = 1, 2, \dots, n$, thuộc tập A được ký hiệu là $a_i \in A$ và phần tử không thuộc - bởi $a_i \notin A$. Số phần tử của một tập hợp được gọi là lũy thừa của tập hợp (đối với tập A ở trên số này bằng n), và đôi khi chúng ta sẽ sử dụng $|A|$ để biểu thị lũy thừa. Với $n = 0$, tập được gọi là *rỗng* và được ký hiệu là \emptyset . Thông thường,

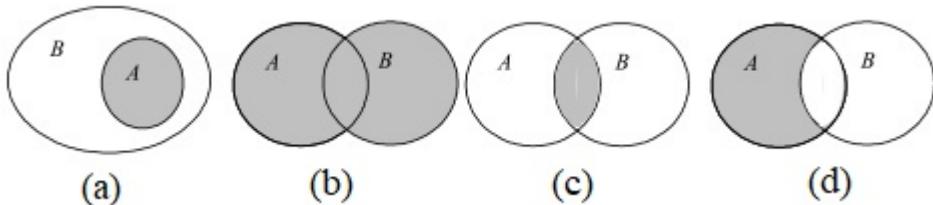
các tập hợp được biểu diễn bằng biểu đồ Venn (xem Hình 1.1.): Mỗi tập hợp A được biểu diễn dưới dạng một vùng đóng và các phần tử (a, b, \dots) - là các điểm nằm trong, hoặc bên ngoài nó, tùy thuộc vào việc chúng có thuộc bộ hay không.



Hình 1.1. Sơ đồ Cen

Có nhiều khái niệm, tính chất và phép toán thống nhất trong một lý thuyết tập hợp được xây dựng tốt [Dilova, Stoyanov-1973]. Chúng ta sẽ đề cập ngắn gọn thêm một số trong số đó, sẽ cần trong tài liệu sau.

Định nghĩa 1.1. Nếu tất cả các phần tử của một tập hợp A đã cho là phần tử của một tập hợp B khác, thì A được gọi là *tập hợp con* của B . Điều này được ký hiệu là $A \subseteq B$ (xem Hình 1.2.). Khi biết thêm rằng B không trùng với A thì tập A được gọi là *tập con thích hợp* (thực) của B . Trong trường hợp này chúng ta sẽ sử dụng kí hiệu $A \subset B$.



Hình 1.2. Tập hợp con (a), tập hợp hợp (b), tập giao (c) và tập hiệu (d) của các tập hợp.

Định nghĩa 1.2. Tập hợp C được gọi là *hợp* của tập A và B nếu nó bao gồm tất cả các phần tử a sao cho $a \in A$ hoặc $a \in B$. Viết $C = A \cup B$.

Định nghĩa 1.3. Giao $C = A \cap B$ của hai tập hợp A và B được gọi là tập hợp C , gồm tất cả các phần tử đồng thời thuộc A và B .

Định nghĩa 1.4. Hiệu $C = A \setminus B$ của tập A và B được gọi là tập C , gồm tất cả các phần tử thuộc A nhưng không thuộc B .

Định nghĩa 1.5. Một tập hợp được gọi là *hữu hạn* nếu nó chứa một số hữu hạn phần tử. Nếu không nó được gọi là *vô hạn*.

Trước khi chúng ta tiếp tục, đây là một vài định nghĩa quan trọng hơn. Khi xem xét một tập hợp, không quan trọng số lần một phần tử xuất hiện, cũng như thứ tự của các phần tử của nó. Do đó các tập hợp sau là tương đương (trùng nhau):

$$\{a, a, b\} = \{a, b, b\} = \{a, b, a\} = \{a, b\} = \{b, a\}$$

Định nghĩa 1.6. Một tập hợp trong đó sự lặp lại của các phần tử được cho phép được gọi là *đa tập hợp*.

Định nghĩa 1.7. Nếu một thứ tự của các phần tử được nhập vào một tập hợp n phần tử, thì đối tượng kết quả được gọi là một n -bộ có thứ tự (danh sách).

Chúng ta sẽ sử dụng dấu ngoặc tròn thay vì dấu ngoặc nhọn để biểu thị n -bộ có thứ tự. Ví dụ, ba thứ tự (a, b, c) khác với ba thứ tự (a, c, b) , v.v.

Bài tập

- **1.1.** Các tập hợp $A = \{1, 2, 4, 5, 7\}$ và $B = \{2, 3, 4, 5, 6\}$ được đưa ra. Xác định các tập hợp: $A \cup B, A \cap B, A \setminus B, B \setminus A$.
- **1.2.** Hai tập hợp A và B đã cho và biết rằng $A \cap B = A$. Bạn có thể nói gì về tập hợp B ?
- **1.3.** Toán tử hiệu đối xứng \oplus của A và B được xác định như sau: $A \oplus B = (A \cup B) \setminus (A \cap B)$. Xác định hiệu đối xứng của các tập $A = \{1, 2, 4, 5, 7\}$ và $B = \{2, 3, 4, 5, 6\}$.
- **1.4.** Một phép toán \bullet được gọi là đối xứng nếu $A \bullet B = B \bullet A$. Phép toán nào sau đây là đối xứng: $A \cup B, A \cap B, A \setminus B, B \setminus A, A \oplus B$?

1.1.2. Tập hợp số

Các con số là cơ sở cho tất cả các loại tính toán toán học, cũng như các thuật toán là cơ sở của tin học máy tính. Theo trình tự thời gian (ban đầu chỉ có 3 "số" được dùng để đếm: một, hai và nhiều), chúng được coi là hình thức đầu tiên của tư duy trừu tượng và nhiều người coi chúng như một thú giật kỳ diệu và đặc biệt.

Các con số có liên quan đến hầu hết mọi thuật toán và chương trình. Đó là lý do tại sao các vấn đề liên quan đến việc trình bày, lưu trữ và sử dụng máy tính của họ là vô cùng quan trọng.

Định nghĩa 1.8. Tập hợp các *số tự nhiên* (chúng ta sẽ ký hiệu là \mathbb{N}) chứa các số mà chúng ta đếm được: 0, 1, 2, 3,

Có tranh cãi về việc số 0 có nên được coi là một số tự nhiên hay không. Nó trừu tượng hơn các số tự nhiên khác và bị thiếu trong nhiều hệ thống số cũ: ví dụ, trong hệ thống số La Mã, việc biểu diễn các số bắt đầu bằng một. Mặt khác, trong toán học rời rạc, việc giới thiệu \mathbb{N} thường được thực hiện từ đầu. Điều này là do khi làm việc với các tập hợp n phần tử, thường phải đưa tập hợp rỗng vào định nghĩa.

Định nghĩa 1.9. Định nghĩa 1.9. Tập hợp các *số nguyên* \mathbb{Z} bao gồm: ..., -3, -2, -1 (số nguyên âm), 0 (số không), 1, 2, 3, ... (số nguyên dương).

Trong bộ nhớ máy tính, chúng được biểu diễn dưới dạng một chuỗi các bit - mã chuyển tiếp, mã ngược hoặc mã bổ sung [Shishkov-1995]. Ngày nay, mã bổ sung thường được sử dụng nhiều nhất, vì nó dễ dàng thu gọn vào phép cộng, thuận tiện cho việc thực hiện trên máy tính.

Các trình biên dịch khác nhau của các loại C và phạm vi định nghĩa của chúng có thể khác nhau. ANSIC (American National Standards Institute [ANSIC]) không xác định các phạm vi cụ thể (Bảng 1.1 làm ví dụ cho thấy các phạm vi cho Borland C cho DOS), mà chỉ xác định các mối quan hệ, ví dụ:

$$|\text{short}| \leq |\text{int}| \leq |\text{long}|$$

Theo mặc định, kích thước của **int** là một từ máy (điều này giải thích tại sao loại DOS là 2 byte và loại Windows là 4).

Một cách chuẩn để nhận giá trị lớn nhất của một loại, chẳng hạn như **unsigned**, là dựa trên biểu diễn bên trong của các số trong mã bổ sung: (**unsigned**)(-1)

Loại	Giá trị	Kích thước
char	-128, ..., 127	8 bit có dấu
unsigned char	0, ..., 255	8 bit không dấu
short int	-32768, ..., 32767	16 bit có dấu
int	-32768, ..., 32767	16 bit có dấu
long int	-2147483648, ..., 2147483647	32 bit có dấu
unsigned short int	0, ..., 65535	16 bit không dấu
unsigned long int	0, ..., 4294967295	32 bit không dấu

Bảng 1.1. Các kiểu số nguyên trong Borland C dành cho DOS.

Ở cột giữa của Bảng 1.1. phạm vi giá trị có thể chấp nhận một biến của kiểu tương ứng được đưa ra. Mỗi kiểu được đặc trưng bởi một số nguyên tối đa và tối thiểu mà nó có thể lưu trữ. Bất kỳ phép gán nào vượt quá giá trị tối đa (hoặc tối thiểu) của kiểu được gọi là tràn và có thể dẫn đến kết quả thảm hại cho từng thuật toán và chương trình.

Định nghĩa 1.10. Số hữu tỉ là những số thuộc loại $\frac{p}{q}$, trong đó p và q là các số nguyên và q là số dương. Tập hợp các số hữu tỉ được kí hiệu là \mathbb{Q} .

Định nghĩa 1.11. Số thực là những số có thể viết dưới dạng:

$$x = n + 0, d_1 d_2 d_3 \dots,$$

với n là một số nguyên và d_i là các chữ số thập phân từ 0 đến 9.

Các số $0, d_1 d_2 d_3 \dots$ được gọi là số thập phân. Trình bày ở dạng trên có nghĩa là bất đẳng thức

$$n + \frac{d_1}{10} + \frac{d_2}{100} + \dots + \frac{d_k}{10^k} \leq x \leq n + \frac{d_1}{10} + \frac{d_2}{100} + \dots + \frac{d_k}{10^k} + \frac{1}{10^k}$$

với mọi $k \in \mathbb{N}, k \geq 0$.

Lưu ý rằng dãy chữ số d_i có thể là vô hạn. Điều này có thể xảy ra đối với cả số hữu tỉ và số vô tỉ (tức là số không hữu tỉ). Ví dụ, số $1/3 = 0,333333\dots$. Ở đây số 3 được lặp lại vô hạn và được gọi là chu kỳ của phân số. Viết thêm $1/3 = 0,(3)$ và đọc "không nguyên và ba trong khoảng thời gian". Dấu chấm có thể dài hơn một chữ số, ví dụ $1/7 = 0,(142857)$.

Ví dụ về một số thực vô tỷ và không tuần hoàn, tức là không được biểu diễn chính xác dưới dạng p/q ($p, q \in \mathbb{N}, q > 0$), là tỷ số giữa chu vi hình tròn với đường kính của nó - hằng số π :

$$\pi = 3,141592653\dots$$

Giống như bất kỳ số thực nào, π có thể được tính gần đúng bằng một số hữu tỉ - ví dụ $355/113$, sẽ xác định nó với độ chính xác sau dấu thập phân, nhưng sau một số chữ số nhất định (đối với ví dụ đã chọn, nó là 6 chữ số sau dấu dấu thập phân) độ chính xác này bị mất. Một giá trị gần đúng hữu ích khác là $22/7$.

Trong bộ nhớ máy tính, số thực về mặt lý thuyết có thể được biểu diễn theo ba cách: cố định, tự nhiên và dấu phẩy động [Shishkov-1995]. Trong thực tế, số dấu phẩy động được sử dụng phổ biến nhất theo tiêu chuẩn IEEE ((Institute of Electrical & Electronics Engineers). Các nhận xét tương tự cũng áp dụng cho các kiểu thực cũng như kiểu số nguyên. Trong Bảng 1.2. phạm vi của các loại thực tế trong Borland C cho DOS được hiển thị

Loại	Giá trị	Kích thước
float	$3,4 \cdot 10^{-38}, \dots, 3,4 \cdot 10^{38}$	32 bit
double	$1,7 \cdot 10^{-308}, \dots, 1,7 \cdot 10^{308}$	64 bit
long double	$3,4 \cdot 10^{-4932}, \dots, 1,1 \cdot 10^{4932}$	80 bit

Bảng 1.2. Các kiểu số th trong Borland C.

Khi làm việc với số thực, phải cẩn thận để tránh hiện tượng tràn và mất độ chính xác sau dấu thập phân (underflow): Khi biểu diễn với một số bit cố định, độ chính xác bị mất khi làm tròn (kết quả là chúng ta chỉ có thể sử dụng một số lượng bit giới hạn). Khi biểu

diễn một số thực dưới dạng phân số hữu tỉ, độ chính xác bị mất đi so với tính gần đúng trong biểu diễn: $1/3$ là phân số thập phân vô hạn tuần hoàn và không thể biểu diễn chính xác bằng một số thực hữu hạn.

Do đó, đối với mỗi kiểu dữ liệu biểu diễn một số thực, tồn tại số ϵ nhỏ nhất ($\epsilon > 0$) sao cho bất kỳ số nào (hoặc kết quả của một phép tính số học) nhỏ hơn ϵ về giá trị tuyệt đối được làm tròn thành 0.

Bài tập

► 1.5. Ch minh rằng rằng các số thực có thể được biểu diễn dưới dạng số dâu phẩy động thực tiêu chuẩn trên thực tế là một tập con hữu hạn của các số hữu tỉ.

1.1.3. Phép chia hết và chia có dư

Gọi m và n là các số nguyên, $m \neq 0$. Khi đó tồn tại các số nguyên q và r ($0 \leq r < m$) sao cho $n = q.m + r$. Số q được gọi là *thương* của phép chia số nguyên n/m , và r được gọi là *phần dư*. Nếu phần dư r của phép chia số nguyên là 0, ta nói rằng m chia cho n (n là bội của m) và viết $m|n$. Trong ngôn ngữ C, phép chia với phần nguyên và phần dư (khi làm việc với số nguyên) được thực hiện với sự trợ giúp của các phép toán / và %. Để tránh "nhầm lẫn", chúng ta sẽ sử dụng hai ký hiệu này trong các văn bản toán học:

$$\begin{aligned} q &= n/m; \\ r &= n \% m; \end{aligned}$$

Định nghĩa 1.12. Khi $(n - m)\%z = 0$, ta nói rằng n có thể so sánh với m modulo z và viết $n \equiv m \pmod{z}$.

Phép chia cho thương và dư có thể dùng để tìm số chữ số của số tự nhiên n . Thuật toán như sau: Chia liên tiếp (nếu có thể) một số nguyên n cho 10. Rõ ràng, với mỗi phép chia như vậy, các chữ số của n giảm đi một. Như vậy, số chữ số của n được xác định bằng số phép chia ta thực hiện cho đến khi n bằng 0

Chương trình 1.1. Số chữ số (101digits.c)

```
#include <stdio.h>
```

```

unsigned m, n = 424267;

int main(void) {
    unsigned digits;
    m = n;
    for (digits = 0; n > 0; n /= 10, digits++);
    printf("So nhung chu so cua %u la %u\n", m, digits);
    getchar();
    return 0;
}

```

Bài tập

- ▷ 1.6. Tìm thương và dư của phép chia m cho n nếu (m, n) là: $(7, 3), (-7, 3), (7, -3), (-7, -3), 3, 7), (-3, 7), (3, -7), (-3, -7)$.
- ▷ 1.7. Đối với các mục tiêu đã cho, m và n ($m \neq 0$) có gắng chứng minh sự tồn tại và tính duy nhất của biểu diễn $n = q.m + r, 0 \leq r < m, (q, r$ số nguyên). [Siderov-1995]
- ▷ 1.8. Đề xuất giải thuật tìm số chữ số của một số thực. Những vấn đề gì phát sinh?

1.1.4. Tính tổng và tích

Các số a_1, a_2, \dots, a_n được cho trước. Một trong ba ký hiệu sau đây thường được sử dụng nhất để biểu thị tổng của chúng $S = a_1 + a_2 + \dots + a_n$:

$$S = \sum_{i=1}^n a_i, S = \sum_{1 \leq i \leq n} a_i, \text{ hoặc } c$$

Có thể cho một hàm $R(x)$ tạo ra các giá trị của i - khi đó tổng được viết dưới dạng:

$$S = \sum_{i:R(x)} a_i,$$

Hàm rút gọn sau đây của trong ngôn ngữ C tìm tổng S_n của n số tự nhiên đầu tiên:

Chương trình 1.2. Tính tổng cách 1 (102sum1.c)

```
#include <stdio.h>
unsigned sum(unsigned n)
{ unsigned i, s = 0;
  for (i = 1; i <= n; i++) s += i;
  return s;
}
int main(void) {
  unsigned s;
  s=sum(1000);
  printf("Tong la %u\n",s);
  getchar();
  return 0;
}
```

Tương tự ta có thể tính tổng mà các phần tử nằm trong mảng n thành phần.

Chương trình 1.3. Tính tổng cách 2 (103sum1.c)

```
#include <stdio.h>
int sum(unsigned n)
{ unsigned i;
  int a[n];
  for (i = 0; i < n; i++) a[i] = i;
  int s = 0;
  for (i = 0; i < n; i++) s += a[i];
  return s;
}

int main(void) {
  unsigned s;
  s=sum(100);
  printf("Tong la %u\n",s);
  getchar();
  return 0;
}
```

Trong ví dụ đầu tiên, chu kỳ là dư thừa - tổng có thể được tìm

trực tiếp bằng công thức cấp số cộng:

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

Trong chương trình thứ hai - một công thức trực tiếp như vậy không thể tồn tại, bởi vì các phần tử của mảng có thể là các số nguyên tùy ý.

Trường hợp tổng lồng nhau, ta sử dụng chu kỳ lồng nhau

$$\begin{aligned} \sum_{j=1}^n \sum_{i=1}^m a_j b_i &= a_1 b_1 + a_1 b_2 + \cdots + a_1 b_n + \cdots + a_n b_1 + \cdots + a_n b_n \\ &= a_1(b_1 + \cdots + b_m) + a_2(b_1 + \cdots + b_m) + \cdots + a_n(b_1 + \cdots + b_m) \\ &= (a_1 + \cdots + a_n)(b_1 + \cdots + b_m) \\ &= \sum_{j=1}^n a_j \sum_{i=1}^m b_i. \end{aligned} \tag{1.1}$$

Các đầu vào như vậy có thể được tính toán bằng cách sử dụng các chu kỳ đầu vào:

Chu kỳ lồng nhau

```
unsigned i, j;
int result = 0;
for (j = 1; j <= n; j++)
    for (i = 1; i <= m; i++)
        result += a[i] * b[j];
```

Hai thuộc tính thú vị khác hợp lệ với số tổng là:

$$\sum_{i:R(x)} \sum_{j:S(x)} a_{ij} = \sum_{j:S(x)} \sum_{i:R(x)} a_{ij} \tag{1.2}$$

$$\sum_{R(x)} a_i + \sum_{S(x)} a_i = \sum_{S(x)||R(x)} a_i + \sum_{S(x)\&&R(x)} a_i. \tag{1.3}$$

Ở đây với $S(x)||R(x)$ và $S(x)\&&R(x)$, chúng ta đã biểu thị tương ứng, liên hợp và giao điểm của các giá trị được tạo bởi các hàm S và R , có thể chấp nhận tham số i và j .

Trong tích các số $P = a_1.a_2.a_3.....a_n$ ký hiệu toán học được sử dụng là:

$$P = \prod_{i=1}^n a_i, P = \prod_{1 \leq i \leq n} a_i, \text{ hoặc } P = \prod_{i=1..n} a_i$$

Một hàm của ngôn ngữ C để tìm tích các phần tử của mảng $a[]$, với n phần tử, có thể có dạng sau:

Tính tích của một mảng số trong C.

Chương trình 1.4. Tính tích (104mult.c)

```
#include <stdio.h>
int mult(unsigned n)
{
    unsigned i;
    int a[n];
    for (i = 0; i < n; i++) a[i] = i+1;
    int s = 1;
    for (i = 0; i < n; i++) s = s*a[i];
    return s;
}
int main()
{
    unsigned s;
    s = mult(13);
    printf("Tich la %u\n", s);
    getchar();
    return 0;
}
```

Bài tập

- ▷ 1.9. Tìm ra công thức tổng của một cấp số cộng.
- ▷ 1.10. Chứng minh các tính chất (1.1), (1.2) và (1.3) của tổng.
- ▷ 1.11. Công thức và chứng minh cho một tính chất sản phẩm tương tự như (1.1), (1.2) và (1.3).
- ▷ 1.12. Sử dụng các thuộc tính (1.1), (1.2) và (1.3), kiểm tra tính hợp lệ của đẳng thức:

$$\sum_{i=1..n} a_i + \sum_{i=n..m} a_i = \sum_{i=1..m} a_i + a_n, \quad 1 \leq n \leq m.$$

1.1.5. Tính lũy thừa, logarit và căn

Định nghĩa 1.13. Nếu x là số thực và y là số tự nhiên thì tung độ được xác định như sau:

$$x^y = x \cdot x \cdot \dots \cdot x (y \text{ lần})$$

Khi $y < 0$ thì $x^y = 1/x^{-y}$.

Các thuộc tính sau là hợp lệ ($x \neq 0$):

$$x^y = x^{y-1} \cdot x \cdot x^y = x^{y+1} / x \cdot x^{y_1+y_2} = x^{y_1} \cdot x^{y_2} x^{y_1+y_2} = (x^{y_1})^{y_2}$$

Việc thực hiện đơn giản để tính x^y là bằng cách thực hiện các phép nhân liên tiếp y :

Chương trình 1.5. Tính lũy thừa (105power.c)

```
#include <stdio.h>
float power(float x, unsigned y)
{
    float res = x;
    unsigned i;
    for (i = 1; i < y; i++) res *= x;
    return res;
}
int main(void) {
    float s;
    s=power(3,7);
    printf("Lũy thừa là %f\n",s);
    getchar();
    return 0;
}
```

Sau đó (xem 7.5.) Chúng ta sẽ xem xét một số cách khác, thú vị hơn và hiệu quả hơn để tìm x^y .

Nếu $x^n = y$ (n là số tự nhiên, $n > 1$) thì x được gọi là căn bậc n của y và viết $x = \sqrt[n]{y}$. Phép toán trong đó gốc thứ y của y nhận được từ một y cho trước được gọi là *phptnhcn*. Đôi với $n = 2$, thay vì căn bậc hai của y , hãy nói *căn bậc hai* hoặc chỉ *căn y* và viết \sqrt{y} . Sử dụng thao tác lấy căn, việc tăng một số có thể được xác định một cách hợp lý:

$$x^{\frac{p}{q}} = \sqrt[q]{x^p}.$$

Trường hợp thú vị nhất là khi x và y là các số thực ($x > 1$). Gọi y được biểu diễn dưới dạng $y = d, d_1d_2d_3\dots$ và xét ràng buộc sau:

$$x^{d + \frac{d_1}{10} + \frac{d_2}{100} + \dots + \frac{d_k}{10^k}} \leq x^y \leq x^{d + \frac{d_1}{10} + \frac{d_2}{100} + \dots + \frac{d_k}{10^k} + \frac{1}{10^k}}.$$

Rõ ràng, nó định nghĩa x^y là một số thực duy nhất - chúng ta có thể nhận được bất kỳ số chữ số nào của nó sau dấu thập phân, miễn là chúng ta đã chọn một số k đủ lớn.

Chúng ta sẽ xem xét một chức năng quan trọng khác sẽ cần thiết trong tài liệu sau này, đặc biệt là trong phân tích độ phức tạp của một thuật toán. Phương trình $y = b^x$ với $b \neq 1, b > 0, y > 0$ có nghiệm duy nhất là x . Nghiệm này được gọi là *logarit của y tại cơ số b* và được ký hiệu là $\log_b y$. Dưới đây là một số tính chất của hàm logarit ($x > 0, y > 0, b > 0, b \neq 1, c > 0, c \neq 1$):

$$x = b^{\log_b x} = \log_b b^x \quad (1.4)$$

$$\log_b(xy) = \log_b x + \log_b y \quad (1.5)$$

$$\log_b x^y = y \log_b x \quad (1.6)$$

$$\log_b x = \frac{\log_c x}{\log_c b} \quad (1.7)$$

Ở phần sau của cuốn sách, chúng ta thường sẽ bỏ qua cơ số của logarit khi nó là 2 và chúng ta sẽ chỉ đơn giản là viết $\log x$ thay vì $\log_2 x$. Chúng ta cũng sẽ sử dụng ký hiệu $\ln x$ để biểu thị một logarit tự nhiên là $\log_e x$: dựa trên số Hepper $e = 2.71828\dots$ (xem 1.1.6.).

1.1.6. Bài tập

▷ 1.13. Để chứng minh các thuộc tính nêu trên của mức độ, bắt đầu từ định nghĩa.

▷ 1.14. Chứng minh các tính chất (1.4), (1.5) và (1.6) của lôgarit, bắt đầu từ định nghĩa.

1.1.7. Giai thừa và hồi qui

Thừa số của $n, n \in \mathbb{N}$ (viết $n!$) Là tích của các số tự nhiên từ 1 đến n :

$$n! = 1.2.\dots.n = \prod_{i=1}^n$$

theo định nghĩa $0! = 1$.

Trong ngôn ngữ C để tính $n!$ không đặc biệt khó khăn:

Chương trình 1.6. Giai thừa (106factorial.c)

```
#include <stdio.h>
const unsigned n = 10;
unsigned long factoriel(unsigned n)
{
    unsigned i;
    unsigned long r = 1;
    for (i = 2; i <= n; i++) r *= i;
    return r;
}
int main()
{
    printf("%u! = %lu\n", n, factoriel(n));
    return 0;
}
```

Đối với một số hàm toán học và dãy số, định nghĩa lặp lại tương ứng thuận tiện và rõ ràng hơn nhiều. Điều này có nghĩa là xác định hàm bằng chính nó hoặc tính toán từng phần tử kế tiếp của chuỗi các giá trị của hàm, sử dụng các giá trị của hàm trước đó. Trong trường hợp của chúng ta, định nghĩa lặp lại của một giai thừa trông giống như sau:

$$n! = \begin{cases} 1, & n = 0 \\ n(n-1)!, & n > 0 \end{cases}$$

Tương tự với giai thừa cho tổng của n số tự nhiên S_n đầu tiên, chúng ta thu được định nghĩa truy hồi:

$$S_n = \begin{cases} 0, & n = 0 \\ S_{n-1} + n, & n > 0 \end{cases}$$

Chúng ta sẽ thấy ở phần sau (xem 1.2.) Điều đó để tính toán cho mỗi hàm toán học lặp lại, một *hàm đệ quy* tương ứng của C có thể được viết.

Bài tập

- 1.15. Đề xuất công thức truy hồi để nâng x thành lũy thừa y ($x \in \mathbb{R}, y \in \mathbb{N}$).

► **1.16.** Đề xuất công thức tìm ước số chung lớn nhất của hai số tự nhiên.

1.1.8. Ma trận

Ma trận là một bảng số hình chữ nhật (nói chung là một bảng hình chữ nhật gồm các đối tượng ngẫu nhiên).

Các số m và n xác định số chiều ($m \times n$) của ma trận: tức là chứng tỏ rằng nó bao gồm m hàng và n cột ($n \geq 1, m \geq 1$). Khi $m = n$ ma trận được gọi là *hình vuông*. Mỗi phần tử a_{ij} của ma trận được đặc trưng bởi một chỉ số kép: số đầu tiên trong đó xác định số hàng và số thứ hai - số bậc thang nơi nó nằm (xem Hình 1.3.).

a_{11}	a_{12}	\dots	a_{1n}
a_{21}	a_{22}	\dots	a_{2n}
...			
a_{m1}	a_{m2}	\dots	a_{mn}

$A_{m \times n} =$

Hình 1.3. Ma trận $m \times n$

Ma trận được sử dụng rộng rãi trong đại số, trong giải hệ phương trình, trong hoạt hình máy tính, trong đó ma trận có kích thước $2 \times 2, 3 \times 3, 4 \times 4$ được sử dụng để dịch và quay các đối tượng, để đạt được các hiệu ứng đồ họa khác nhau (chẳng hạn như phôi cảnh xem trong địa hình ba chiều), v.v. Việc sử dụng ma trận trong các trường hợp này và các trường hợp khác dẫn đến các thuật toán hiệu quả hơn đáng kể và do đó, năng suất cao hơn [Ayres-1962].

Trong ngôn ngữ C một bảng chữ nhật ta có thể biểu diễn như một mảng

int A[m][n];

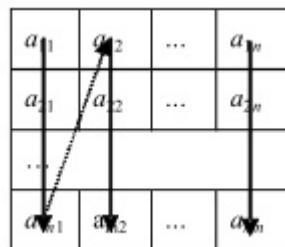
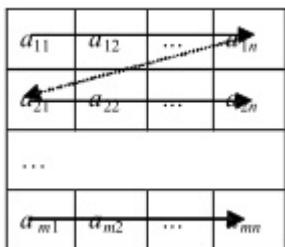
Trường hợp tổng quát hơn cũng có thể xảy ra khi các phần tử của ma trận là các đối tượng tùy ý của một số kiểu **struct** data được xác định trước, ví dụ:

Loại dữ liệu ma trận

```
struct data {
    int a;
    int b;
    ...
} A[m][n];
```

Nhập vào và in ra một ma trận

Phần tử ma trận có thể được đọc/in theo nhiều cách khác nhau. Hai đơn giản nhất là ma trận *hàng* hoặc *cột* (Hình 1.1.1d).



Hình 1.4. Thu thập thông tin các phần tử của ma trận: (a) theo hàng và (b) theo cột.

Nhập dữ liệu ma trận

```
//Nhập theo hàng
for (i = 0; i < m; i++)
    for (j = 0; j < n; j++) scanf("%d", &A[i][j]);

//Nhập theo cột
for (i = 0; i < m; i++)
    for (j = 0; j < n; j++) scanf("%d", &A[j][i]);

// In theo hàng
for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) printf("%.3d", A[i][j]);
    printf("\n");
```

Tổng hai ma trận

Tổng của hai ma trận $A_{m \times n}$ và $B_{m \times n}$ là ma trận thứ ba $C_{m \times n}$ sao cho $c_{ij} = a_{ij} + b_{ij}$ (với $i = 1, 2, \dots, m, j = 1, 2, \dots, n$), xem Hình 1.5.

a_{11}	a_{12}	\dots	a_{1n}	+	b_{11}	b_{12}	\dots	b_{1n}	=	$a_{11} + b_{11}$	$a_{12} + b_{12}$	\dots	$a_{1n} + b_{1n}$
a_{21}	a_{22}	\dots	a_{2n}		b_{21}	b_{22}	\dots	b_{2n}		$a_{21} + b_{21}$	$a_{22} + b_{22}$	\dots	$a_{2n} + b_{2n}$
\dots					\dots					\dots			
a_{m1}	a_{m2}	\dots	a_{mn}		b_{m1}	b_{m2}	\dots	b_{mn}		$a_{m1} + b_{m1}$	$a_{m2} + b_{m2}$	\dots	$a_{mn} + b_{mn}$

Hình 1.5. Tổng các ma trận.

Tổng hai ma trận

```

for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
        C[i][j] = A[i][j] + B[i][j];
    
```

Tích hai ma trận

Tích của hai ma trận $A_{m \times n}$ và $B_{n \times p}$ là ma trận thứ ba $C_{m \times p}$ mà:

$$c_{ij} = \sum_{k=1}^n (a_{ik}b_{kj}), \text{ với mỗi } i = 1, 2, \dots, m \text{ và } j = 1, 2, \dots, p.$$

Thực hiện bên dưới, áp dụng trực tiếp công thức từ định nghĩa và thực hiện phép nhân đơn giản $m.p.n$. Nếu $n > m$ và $n > p$ thì số phép nhân nguyên tố bị giới hạn bởi n^3 .

Tích hai ma trận

```

for (i = 0; i < m; i++)
    for (j = 0; j < p; j++) {
        C[i][j] = 0;
        for (k=0;k<n;k++)
            C[i][j] += A[i][k]*B[k][j];
    }
    
```

Tích của ma trận có nhiều ứng dụng - trong việc giải các hệ phương trình, trong thống kê, trong lý thuyết đồ thị và các ứng dụng khác. Có một phương pháp để nhân ma trận số, thực hiện phép nhân đơn giản $n^{\log 7}$ ($\approx n^{2.81}$), đồng thời thực hiện các phép cộng bổ sung. Phương pháp được gọi là *phương pháp Strassen* để nhân

nhanh các ma trận sẽ được xem xét trong 7.6., Như một minh chứng của kỹ thuật *chia và trị*.

Bài tập

► 1.17. Một ma trận được cho trước `unsigned a[MAX][MAX]`. Viết một hàm

`void fillMatrix (unsigned a[][MAX], unsigned n)`,

điền vào các phần tử của `a[][]` như sau:

0	20	19	17	14
1	0	18	16	13
2	5	0	15	12
3	6	8	0	11
4	7	9	10	0

► 1.18. Một ma trận không dấu được đưa ra `a[MAX][MAX]`. Viết một hàm hiển thị phần tử xoắn ốc của nó, ví dụ với $n = 5$, chúng ta có:

1	16	15	14	13
2	17	24	23	12
3	18	25	22	11
4	19	20	21	10
5	6	7	8	9

1.1.9. Tìm số chữ số của một tích

Bài toán: Cho trước các số nguyên a_1, a_2, \dots, a_n . Tìm số chữ số của tích $P = a_1.a_2.\dots.a_n$.

Đặc biệt, nếu $a_i = i$, với $i = 1, 2, \dots, n$, thì hãy tìm số chữ số của $P = 1, 2, \dots, n = n!$.

Một giải pháp rõ ràng là thực hiện phép nhân và sử dụng thuật toán 1.1.1, tìm số chữ số của kết quả. Đối với ví dụ đã chọn, điều này có nghĩa là tính toán $n!$. Hàm $n!$ tuy nhiên, nó đang phát triển quá nhanh, trong khi số chữ số đang tăng chậm hơn nhiều. $10!$ là 3628.800, và con số của nó chỉ là bảy. Ở tuổi 20! kết quả vượt quá giá trị lớn nhất cho một kiểu số nguyên trong C.

Chúng ta sẽ áp dụng một thuật toán hiệu quả hơn dựa trên mối quan hệ giữa số chúng ta nhân và số chữ số trong tích kết quả:

Số chữ số của P bằng $[1 + \sum_{i=1}^n \log_{10} a_i]$, trong đó $[x]$ biểu thị số nguyên lớn nhất nhỏ hơn hoặc bằng x .

Để giải thích tính hợp lệ của công thức trên, chúng ta sẽ chú ý đến thực tế là số chữ số của mỗi số nguyên P bằng $[1 + \log_{10}(P)]$, cũng như tính chất của hàm logarit:

$$\log P = \log(a_1.a_2....a_n) = \log a_1 + \log a_2 + \dots + \log a_n.$$

Sau đây là thực hiện trên ngôn ngữ C để tìm các chữ số của $n!$

Chương trình 1.7. Số chữ số của tích (110digitsnf.c)

```
#include <stdio.h>
#include <math.h>
const unsigned long n = 123;

int main(void)
{ float digits = 0;
  unsigned i;
  for (i = 1; i <= n; i++) {
    digits += log10(i);
  }
  // Số [x] đưa ra số nguyên dài
  printf("So chu so cua %lu! la %lu\n", n,
         (unsigned long) digits + 1);
  getchar();
  return 0;
}
```

Bài tập

- ▷ 1.19. Chứng minh rằng với mọi số tự nhiên P có số chữ số thập phân của nó bằng $[1 + \log_{10}(P)]$.

1.2. Số nguyên tố

Số nguyên tố đã được xem xét trong toán học từ thời cổ đại và gắn liền với nhiều vấn đề thú vị vượt xa biên giới của nó. Trong khoa học máy tính, chúng được sử dụng trong mã hóa, lưu trữ, v.v.

Định nghĩa 1.14. Một số tự nhiên được gọi là đơn giản nếu không có ước nào khác ngoài 1 và chính nó, và số 1 không được coi là số nguyên tố. Nếu nó không đơn giản, nó được gọi là hợp chất.

Dãy số nguyên tố bắt đầu như sau:

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, \dots$$

Người ta đã chứng minh (Euclid - 300 TCN) rằng các số nguyên tố là vô số.

Như đã đề cập, một trong những ứng dụng của số nguyên tố là mã hóa dữ liệu được truyền qua mạng "không an toàn" (ví dụ: các giao dịch tài chính trên Internet). Một số thuật toán để mã hóa thông tin được truyền (ví dụ RSA [RSA-78]) sử dụng tích các số nguyên tố lớn. Để "phá vỡ" kênh truyền tải thông tin như vậy, bản thân các số nguyên tố phải được biết đến chứ không chỉ là tích của chúng. Nếu nhìn vào số 55, chúng ta có thể đoán ngay nó phân hủy như thế nào - là tích của 5 và 11. Đối với số 4853, chúng ta sẽ khó nhớ lại các ước số đơn giản của nó (211 và 23), nhưng chúng ta có thể biên dịch một chương trình. mà tìm thấy chúng. Nhiều số nguyên tố lớn hiện được biết đến - ví dụ: nếu chúng ta nhân hai số nguyên tố ngẫu nhiên có 100 chữ số và chỉ có tích kết quả, thì việc khôi phục các số bằng bất kỳ thuật toán nào sẽ mất một thời gian dài không thể chấp nhận được (ví dụ: gấp nhiều lần so với yêu cầu hoàn thành giao dịch ngân hàng).

Khi chúng ta xem xét số lượng các số nguyên tố, một số câu hỏi này sinh:

- Có bao nhiêu số nguyên tố trong khoảng $[a, b]$ cho trước?
- Phần vô cực là số nguyên tố?
- Có công thức tìm số nguyên tố thứ n không?

Nếu ta kí hiệu $\pi(x)$ với tất cả các số nguyên tố không vượt quá một số tự nhiên x thì việc tìm công thức tính $\pi(x)$ chính xác sẽ trả lời được ba câu hỏi trên. Thật không may, một công thức như vậy vẫn chưa tồn tại (và không chắc sẽ được tìm thấy - xem 6.2.), Nhưng có những công thức để tính gần đúng $\pi(x)$, chẳng hạn (hãy nhớ lại rằng $\ln x \equiv \log_e x$):

Định lý 1.1 (cho số nguyên tố). $\pi(x) \simeq \frac{x}{\ln(x-a)}$, trong đó a là hằng số dương tùy ý, nhỏ hơn x . (Giá trị gần đúng nhất đạt được là $a = 1$.)

Hệ quả 1.1. Số nguyên tố thứ n xấp xỉ $[n \cdot \ln(n)]$. Một xấp xỉ tốt hơn đạt được với $[n[(\ln(n) + \ln(\ln n - 1))]]$.

Hệ quả 1.2. Xác suất để một số x là xấp xỉ $\frac{1}{\ln x}$.

Trước khi chuyển sang các thuật toán để kiểm tra xem một số có phải là số nguyên tố hay không, chúng ta sẽ đề cập đến một số tính chất và định lý thú vị hơn cho các số nguyên tố [Số nguyên tố-1] [Số nguyên tố-2]:

Giả thuyết của Goldbach

1. Mọi số nguyên $n > 2$ đều có thể được biểu diễn dưới dạng tổng của hai số nguyên tố.
2. Mọi số nguyên $n > 17$ đều có thể được biểu diễn dưới dạng tổng của ba số nguyên tố khác nhau.
3. Mỗi số nguyên có thể được biểu diễn dưới dạng tổng của tối đa sáu số nguyên tố.
4. Mọi số nguyên lẻ $n > 5$ đều có thể được biểu diễn dưới dạng tổng của ba số nguyên tố.
5. Mỗi số chẵn có thể được biểu diễn dưới dạng hiệu của hai số nguyên tố.

Định lý 1.2. Có vô số số rất nguyên tố loại $n^2 + m^2$ và $n^2 + m^2 + 1$.

Giả thuyết. Có vô số số nguyên tố dạng $n^2 + 1$.

Định lý 1.3 (Operman). Luôn luôn có một số nguyên tố giữa n^2 và $(n+1)^2$.

Bài tập

- 1.20.
1. Tại sao việc tìm một công thức chính xác cho $\pi(x)$ sẽ trả lời được cả ba câu hỏi về số nguyên tố đã lập ở trên?
 2. Viết chương trình kiểm tra các giả thuyết của Goldbach.
 3. Viết chương trình kiểm tra định lý Operman.

1.2.1. Kiểm tra một số có phải là số nguyên tố

Một thuật toán hiển nhiên, hệ quả trực tiếp của định nghĩa, là như sau: chúng ta kiểm tra xem mỗi số trong khoảng $\left[2, \frac{p}{2} - 1\right]$ có chia hết cho p hay không và nếu chúng ta tìm thấy một thì theo đó p là hợp số.

Có một số "công thức" để kiểm tra xem một số có đơn giản hay không, nhưng trong thực tế, chúng không thể áp dụng được, vì việc triển khai chúng đòi hỏi nhiều tài nguyên tính toán hơn so với thuật toán vừa mô tả. Một ví dụ là sau đây

Định lý 1.4 (Wilson). *Số p là số nguyên tố nếu và chỉ khi $(p - 1)! \equiv -1 \pmod{p}$.*

Nó phải tính toán $(p - 1)!$, khó thực hiện hơn nhiều và đòi hỏi nhiều phép tính hơn so với việc thực hiện $\frac{p}{2} - 1$ phép chia trong thuật toán trên.

Dễ dàng thấy rằng không cần thiết phải kiểm tra tất cả các số có đến $\frac{p}{2} - 1$: chỉ cần kiểm tra chia hết cho đến \sqrt{p} (bao gồm) là đủ. Điều này là do bất cứ khi nào p có một ước số x , $x > \sqrt{p}$, thì p được biểu diễn dưới dạng $p = x \cdot y$, $y < \sqrt{p}$, tức là còn một số chia nhỏ hơn \sqrt{p} . Sau đây là cách triển khai thuật toán này:

Chương trình 1.8. Số chữ số (111prime.c)

```
#include <stdio.h>
#include <math.h>

const unsigned n = 23;

// Trả lại 1 thì $n$ là nguyên tố; 0 thì $n$ là hợp số
char isPrime(unsigned n)
{
    unsigned i = 2;
    if (n == 2) return 1;
    while (i <= sqrt(n)) {
        if (n % i == 0) return 0;
        i++;
    }
    return 1;
}
```

```

    }

int main(void) {
    if (isPrime(n))
        printf("So %u la nguyen to.\n", n);
    else
        printf("So %u la hop so.\n", n);
}

```

Chúng ta có thể mở rộng kết quả cuối cùng: để tìm rằng p là số nguyên tố, nó đủ để đảm bảo rằng nó không chia hết cho bất kỳ số nguyên tố nào khác trong khoảng $[2, \sqrt{p}]$ (Chúng ta để lại như một bài tập cho bạn đọc để chứng minh rằng kết luận sau là đúng). Vì vậy, nếu chúng ta có k số nguyên tố đầu tiên (ký hiệu là P_i , với $i = 1, 2, \dots, k$), chúng ta sẽ có thể kiểm tra xem có số nào trong khoảng $[2, (P_k)^2]$. Thì đơn giản. Triển khai trong ngôn ngữ C như sau:

Chương trình 1.9. Kiểm tra số nguyên tố (112preproc.c)

```

#include <stdio.h>
const unsigned n = 25;
// Số lượng số nguyên tố mà ta cần phải tính
#define K 25
unsigned prime[K] = {
    2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
    47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97};

// Kiểm tra có phải chăng là số nguyên tố
// bằng cách kiểm tra có ước số trong danh sách prime[]
char checkprime(unsigned n)
{
    unsigned i = 0;
    while (i < K && prime[i] * prime[i] <= n) {
        if (n % prime[i] == 0) return 0;
        i++;
    }
    return 1;
}

int main(void)
{
    if (checkprime(n))

```

```

    printf("So %u la so nguyen to. \n", n);
else
    printf("So %u la hop so. \n", n);
return 0;
}

```

Bài tập

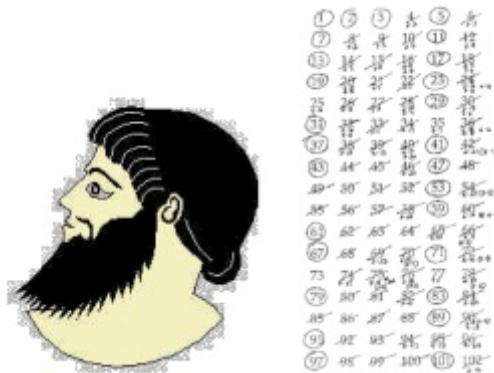
- ▷ 1.21. Viết chương trình kiểm tra xem một số có phải là số nguyên tố hay không, sử dụng định lý Wilson. Có cần thiết phải tính $(p - 1)!$, với điều kiện chúng ta chỉ quan tâm đến modulo p phần dư của nó không?
- ▷ 1.22. Để chứng minh rằng để chứng minh rằng p là số nguyên tố, chỉ cần biết chắc chắn rằng nó không chia hết cho bất kỳ số nguyên tố nào khác trong khoảng $[2, \sqrt{p}]$.

1.2.2. Sàng Eratosten. Tìm số nguyên tố trong khoảng

Bài toán: Tìm tất cả các số nguyên tố nhỏ hơn hoặc bằng một số tự nhiên n . Một cách tiếp cận rõ ràng để giải quyết vấn đề là kiểm tra xem có số tự nhiên nào nhỏ hơn n là đơn giản hay không. Do đó, n lần kiểm tra được thực hiện, và đối với mỗi số k , tối đa k số lần kiểm tra tính chia hết sẽ được yêu cầu.

Với điều kiện có đủ bộ nhớ, chúng ta có thể áp dụng một phương pháp nhanh hơn để tìm các số nguyên tố trong một khoảng gọi là *sàng Eratosthenes*. Phương pháp này được đặt theo tên của người tạo ra nó - Eratosthenes of Siren (275-195 trước Công nguyên) - người đầu tiên dự đoán đường kính chính xác của Trái đất. Ông cũng được biết đến là người đã làm việc tại Thư viện Alexandria trong nhiều năm.

Như tên cho thấy, "sàng" là một phương pháp lập trình loại trừ tất cả các phần tử của một tập hợp hữu hạn mà chúng ta không quan tâm [Rakhnev, Garov, Gavrilov-1995] [Reingold, Nivergelt, Deo-1980]. Nó có thể được minh họa bằng một cái rây lọc spaghetti - nước mà chúng ta muốn "tống khứ" hết đi và spaghetti vẫn còn. Theo cách tương tự, sàng lọc của Eratosthenes "ép" các số tổng hợp, khiến chúng trở nên đơn giản.



Hình 1.6. Eratosthenes và lưới của mình

Trong trường hợp này, ý tưởng của cách tiếp cận này xuất phát từ đoạn trước: một số là đơn giản nếu không có ước số nguyên tố nào khác (ngoại trừ chính nó).

Lưới của Eratosthenes:

Chúng ta viết các số từ 2 đến n liên tiếp:

$$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, \dots, n$$

Chúng ta tìm thấy số đầu tiên chưa được đánh dấu và chưa được đánh dấu – đó là 2. Chúng ta đánh dấu nó, sau đó gạch bỏ mỗi số thứ hai sau nó:

$$(2), 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, \dots, n$$

Tiếp theo, chúng ta tìm lại số đầu tiên chưa được đánh dấu và chưa được đánh dấu: đây là số 3. Chúng ta đánh dấu nó và gạch bỏ tất cả các số trong dãy, bội số của 3:

$$(2), (3), 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, \dots, n$$

Sau đó, đã đến lúc cho số 5 – chúng ta đánh dấu nó và gạch bỏ cứ sau mỗi ngày 5:

$$(2), (3), 4, (5), 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, \dots, n$$

Do đó, tất cả các số tổng hợp đều được "sàng lọc" và chúng ta luôn chắc chắn rằng số i tối thiểu chưa được đánh dấu và chưa được

đánh dấu là đơn giản. Quá trình tiếp tục cho đến khi không có số nào bị gạch chéo hoặc không có dấu - khi đó tất cả các số được đánh dấu là số nguyên tố và tất cả các số bị gạch bỏ đều là hợp số. Dưới đây là sơ đồ chi tiết hơn để thực hiện sàng lọc bằng phương pháp sàng Eratosthenes:

Thuật toán sàng Eratosthenes

1. Khởi tạo mảng sieve[] với các số không. Sau này, khi gạch bỏ một số, chúng ta sẽ viết 1 ở vị trí tương ứng trong mảng i . Chúng ta khởi tạo $i = 2$.
2. Chúng ta tăng i cho đến khi sieve[i] trở thành 0. Khi đó số i là đơn giản và chúng ta in ra.
3. Đánh dấu bằng 1 tất cả các giá trị sieve[k], cho $k = i, 2i, 3i, \dots, (n/i).i$ (tức là tất cả các bội số của i giá trị).
4. Nếu $i \leq n$ thì chúng ta quay lại bước 2, ngược lại thì kết thúc.

Chương trình 1.10. Danh sách số nguyên tố (113sieve.c)

```
#include <stdio.h>
#define MAXN 30000
const unsigned n = 200;
char sieve[MAXN];
//Tìm và in ra số nguyên tố đến $n$
void eratosten(unsigned n)
{ unsigned j, i = 2;
  while (i <= n) {
    if (sieve[i] == 0) {
      printf("%5u", i);
      j = i;
      while (j <= n) {
        sieve[j] = 1;
        j += i;
      }
      i++;
    }
  }
}

int main(void) {
  unsigned i;
  for (i = 0; i < n; i++) sieve[i] = 0;
```

```

eratosthen(n);
return 0;
}

```

Kết quả thực hiện chương trình:

2	3	5	7	11	13	17	19	23	29	31	37	41	43	47
53	59	61	67	71	73	79	83	89	97	101	103	107	109	
113	127	131	137	139	149	151	157	163	167	173	179			
181	191	193	197	199										

Ngoài ra còn có một thuật toán thậm chí còn hiệu quả hơn để tìm kiếm tất cả các số nguyên tố trong một khoảng thời gian. Nó không yêu cầu bộ nhớ (mảng sieve[N]) với kích thước của khoảng, cũng như không cần thiết phải thu thập dữ liệu toàn bộ khoảng ở mỗi bước.

Trong việc triển khai preproc.c từ đoạn trước, chúng ta đã nhập trước mảng số prime[], chứa k số nguyên tố đầu tiên và với sự trợ giúp của nó, chúng ta đã kiểm tra xem một số trong khoảng $[2, (P_k)^2]$ có số nguyên tố.

Bây giờ chúng ta sẽ áp dụng lược đồ đã sửa đổi sau: chúng ta sẽ bắt đầu với một danh sách trống, mà chúng ta sẽ điền tuần tự. Ví dụ, đặt nó vào số nguyên tố đầu tiên 2, chúng ta có thể tìm thấy tất cả các số nguyên tố trong khoảng $[3, 2^2]$ - chẳng hạn như 3 và chúng ta thêm nó vào danh sách. Hơn nữa, đã có các số nguyên tố đến 3, chúng ta có thể tìm thấy tất cả các số nguyên tố trong phạm vi $[4, 3^2]$ - đây là 5 và 7, chúng ta cũng thêm vào danh sách. Chúng ta tiếp tục quá trình này cho đến khi các số nguyên tố được thêm vào prime[] để kiểm tra xem mỗi số trong khoảng $[2, n]$ có phải là số nguyên tố hay không. Sau đây là một ví dụ triển khai:

Kiểm tra một số nguyên tố có nằm trong khoảng $[2, n]$ không?

Chương trình 1.11. Số nguyên tố trong một khoảng (114proc.c)

```

#include <stdio.h>
#define MAXN 10000
//Tìm số nguyên tố đến $n$
const unsigned n = 500;
unsigned primes[MAXN], pN = 0;

```

```

char isPrime(unsigned n)
{ unsigned i = 0;
  while (i < pN && primes[i] * primes[i] <= n) {
    if (n % primes[i] == 0) return 0;
    i++;
  }
  return 1;
}

void findPrimes(unsigned n)
{ unsigned i = 2;
  while (i < n) {
    if (isPrime(i)) {
      primes[pN] = i;
      pN++;
      printf("%5u", i);
    }
    i++;
  }
}

int main(void) {
  findPrimes(n);
  printf("\n");
  return 0;
}

```

Trong trường hợp chúng ta đang tìm n số nguyên tố đầu tiên, phương pháp sàng Eratosthenes cho kết quả rất tốt. Tuy nhiên, khi tìm các số nguyên tố trong khoảng $[a, b]$ cho $a(a >> 1)$ đủ lớn, tốt hơn nên sử dụng một thuật toán khác - một phiên bản cải tiến của thử nghiệm ngay lập tức.

Ba số nguyên tố đầu tiên là 2, 3 và 5. Do đó, bất kỳ số tự nhiên nào cũng có thể được biểu diễn dưới dạng:

$$n = 30q + r, r \in [0..29] \text{ hoặc } r \in [0, \pm 1, \pm 2, \dots, \pm 14, 15].$$

$$30q = 2 \cdot 3 \cdot 5 \cdot q$$

Khi đó trong số 30 số liên tiếp, chỉ có 8 số có thể là số nguyên tố,

tức là chúng ta chỉ cần kiểm tra $4k/15$ số, cụ thể là:

$$30.q \pm 1, 30.q \pm 7, 30.q \pm 11, 30.q \pm 13$$

Phương pháp được đề xuất dẫn đến tăng tốc gấp đôi so với phương pháp xác minh trực tiếp tất cả các con số trong khoảng thời gian xác định.

Bài tập

► 1.23. Thuật toán có thể được cải thiện: ở bước 3) tìm kiếm bắt đầu từ $k = i^2$, trong khi tăng dần với giá trị i được giữ nguyên. Chứng minh cho kết quả này và sửa đổi việc triển khai cho phù hợp. So sánh với bản gốc.

► 1.24. Để chứng minh thuật toán sàng.

► 1.25. Để cải thiện thuật toán tìm kiếm các số nguyên tố trong một khoảng, vì mục đích này, 4 số nguyên tố đầu tiên được xem xét.

1.2.3. Phân tích một số thành tích thừa số nguyên tố

Định lý 1.5 (Định lý cơ bản của số học). *Mọi số tự nhiên P ($P > 1$) đều có thể được biểu diễn (thừa số) một cách duy nhất dưới dạng $P_1^{q_1} \cdot P_2^{q_2} \cdots \cdot P_n^{q_n}$, trong đó $P_1 < P_2 < \dots < P_n$ và P là các số nguyên tố và q_i là các số nguyên dương. [Siderov-1995].*

Dưới đây là một số ví dụ:

$$520 = 2^3 \cdot 5^1 \cdot 13^1$$

$$64 = 2^6$$

$$2345 = 5^1 \cdot 7^1 \cdot 67^1$$

Thuật toán để có được sự phân rã như vậy cần thiết cho một số tác vụ tính toán (ví dụ: Thuật toán 2 của 1.1.5.) Là như sau:

Hãy phân tích số P .

1) Ta đặt $i = 2$.

2) Ta đặt $k = 0$. Trong khi P chia hết cho i , ta thực hiện phép chia và tăng k lên một. Chúng ta chuyển sang 3).

3) Nếu $k > 0$, thì ta thu được số hạng tiếp theo của phân thức - đó là i^k . Chuyển sang 4)

4) Nếu $P > 1$, ta tăng i lên một và trở về 2).

Chúng ta để nó cho người đọc để chứng minh rằng thuật toán đang hoạt động bình thường. Sau đây là nhận thức đầy đủ của C:

Chương trình 1.12. Phân tích số thành tích thừa số nguyên tố (115numdev.c)

```
#include <stdio.h>
//Số để phân tích ra thừa số
unsigned n = 435;

int main(void) {
    unsigned how, i, j;
    printf("%u = ", n);
    i = 1;
    while (n != 1) {
        i++;
        how = 0;
        while ((n % i) == 0) {
            how++;
            n = n / i;
        }
        for (j = 0; j < how; j++) printf("%u ", i);
    }
    printf("\n");
    return 0;
}
```

Bài tập

▷ 1.26. Chứng minh thuật toán được đề xuất để phân tích nhân tử.

1.2.4. Tìm số lượng số không trong kết quả phép nhân

Bài toán: Cho trước một số số nguyên a_1, a_2, \dots, a_n . Chúng ta tìm số lượng các số không mà tại đó tích $P = a_1.a_2.....a_n$.

Như trong 1.1.2, phép nhân là không mong muốn và không phải lúc nào cũng dẫn đến kết quả mong muốn. Để giải quyết vấn đề,

chúng ta sẽ chú ý đến thực tế sau: Các số duy nhất có tích kết thúc bằng 0 là 2 và 5, hoặc tích của bội số của 2 với bội số của 5.

Thuật toán giải quyết vấn đề mà không cần thực hiện phép nhân như sau:

1. Với mỗi $a_i (i = 1, 2, \dots, n)$ ta biểu diễn a_i dưới dạng $a_i = 2^{M_i} \cdot 5^{N_i} \cdot b_i$, $b_i \% 2 \neq 0, b_i \% 5 \neq 0$.
2. Kết quả của sản phẩm sẽ là $P = 2^{\sum_{i=1..n} M_i} \cdot 5^{\sum_{i=1..n} N_i} \cdot c$, (c là hằng số), và số lượng các số không ở cuối sản phẩm sẽ là nhỏ nhất của $\sum_{i=1}^n M_i$ và $\sum_{i=1}^n N_i$.

Ví dụ, đối với chuỗi:

25, 4, 20, 11, 13, 15

sau khi phân hủy chúng ta sẽ nhận được:

$2^0 \cdot 5^2 \cdot 1, 2^2 \cdot 5^0 \cdot 1, 2^2 \cdot 5^1 \cdot 1, 2^0 \cdot 5^0 \cdot 11, 2^0 \cdot 5^0 \cdot 13, 2^0 \cdot 5^1 \cdot 3,$

mà cho 4 số không. Ta có thể kiểm tra tính đúng đắn của kết quả thu được một cách trực tiếp: $25 \cdot 4 \cdot 20 \cdot 11 \cdot 13 \cdot 15 = 4290000$.

Chúng ta sẽ để việc triển khai thuật toán vừa được mô tả như một bài tập cho người đọc. Nó sẽ là một sửa đổi nhỏ của thuật toán đã được xem xét để phân tách số lượng các ước số nguyên tố từ đoạn trước.

Chúng ta sẽ kết thúc với một công thức để tìm số lượng các số không ở cuối $n!$. Số này bằng $\sum_{k=1}^{\lfloor \log_5 n \rfloor} \left\lfloor \frac{n}{5^k} \right\rfloor$. Công thức được suy ra như một hệ quả của sơ đồ tổng quát hơn, sau khi tính xem có bao nhiêu lần các chữ số 2 và 5 tham gia làm ước của n số tự nhiên đầu tiên.

Chương trình 1.13. Tìm số 0 làm phép nhân (116factzero.c)

```
#include <stdio.h>
const unsigned n = 10;
int main(void) {
    unsigned zeroes = 0, p = 5;
    while (n >= p) {
        zeroes += n / p;
        p *= 5;
    }
}
```

```

printf("So luong so 0 cua %u! la %u\n", n, zeroes);
return 0;
}

```

Bài tập

- ▷ 1.27. Để chứng minh thuật toán được đề xuất để tìm số lượng các số không mà tại đó phép nhân kết thúc.
- ▷ 1.28. Để thực hiện thuật toán được đề xuất tìm số lượng các số không tại đó thuật toán kết thúc.
- ▷ 1.29. Biện minh cho công thức tìm số lượng các số không kết thúc bằng $n!$

1.3. Số mersen và số hoàn thiện

1.3.1. Số mersen

Định lý 1.6. Một số nguyên tố được gọi là số nguyên tố Mersenne nếu nó có thể được biểu diễn dưới dạng $2^p - 1$, trong đó p là số nguyên tố.

Hiện tại, 39 giá trị của p đã được biết, trong đó các số $2^p - 1$ là Mersenne:

2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, 2203, 2281, 3217, 4253, 4423, 9689, 9941, 11213, 19937, 21701, 23209, 44497, 86243, 110503, 132049, 216091, 756839, 859433, 1257787, 1398269, 2976221, 3021377, 6972593, 13466917

37 giá trị đầu tiên của p số tương ứng với lũy thừa của 37 số Mersenne đầu tiên. Hai số cuối cùng là của Mersen (lần lượt được phát hiện vào năm 1999 và 2001), nhưng chúng không nhất thiết phải là số 38 và 39, vì chúng được tìm thấy bằng máy tính mà không cần kiểm tra tất cả các số nhỏ hơn. các giá trị của p . nghĩa là, có thể có số Mersenne nhỏ hơn.

Số Mersenne tìm thấy một số ứng dụng: để tìm kiếm các số hoàn hảo, để tìm kiếm các số nguyên tố rất lớn và các ứng dụng khác. Chúng vừa có giá trị thực tế vừa mang giá trị biểu tượng. Khi số nguyên tố thứ 23 của Mersen được tìm thấy vào năm 1963, Trường

Toán học của Đại học Illinois đã rất tự hào về khám phá của mình đến nỗi tất cả các bức thư được gửi kèm theo một con tem bổ sung là " $2^{11213} - 1$ là số nguyên tố" (xem Hình 1.7).



Hình 1.7. $2^{11213} - 1$ là số nguyên tố

Một số số nguyên tố lớn nhất được tìm thấy cho đến nay là Mersenne:

$2^{13466917} - 1$ với 8107892 chữ số (?? số nguyên tố Mersenne thứ 39)

$2^{6972593} - 1$ với 2098960 chữ số (?? số nguyên tố Mersenne thứ 38)

$2^{3021377} - 1$ với 909526 chữ số (số nguyên tố Mersenne thứ 37)

$2^{2976221} - 1$ với 895932 chữ số (số Mersenne thứ 36)

$2^{1398269} - 1$ với 420921 chữ số (số Mersenne thứ 35)

Việc công bố các giải thưởng tiền mặt lớn là một hiện tượng ngày càng phổ biến đối với các kết quả khoa học hoặc thực nghiệm nghiêm túc. Ví dụ, có những dự án phổ biến trên Internet để tìm kiếm số nguyên tố lớn tiếp theo (và rất có thể nó sẽ là số nguyên tố của Mersen - xem bên dưới). Những dự án như vậy được tài trợ bởi các tổ chức nghiên cứu nổi tiếng, và việc tham gia vào một dự án như vậy giống như một cuộc xổ số: người may mắn trúng số nguyên tố thứ 38 của Mersen thì trúng 50.000 USD. Giải thưởng cho những con số tiếp theo bắt đầu từ 250.000 đô la [Số nguyên tố-3].

Một thuật toán hiển nhiên để tìm kiếm các số Mersenne như sau: liên tiếp với mỗi số nguyên tố $p = 2, 3, 5, 7, 11, \dots$ chúng ta kiểm tra xem $M = 2^p - 1$ có đơn giản không. Tuy nhiên, thuật toán này cực kỳ

kém hiệu quả và trong thực tế không cho kết quả khả quan, áp dụng cho số lượng lớn. Chìa khóa để tìm ra các số lớn như vậy là định lý Lucas năm 1870, sau đó được Lehmer sửa đổi, và ứng dụng của nó cũng yêu cầu một chương trình nhân các số nhanh (một phương pháp nhân các số như vậy được tìm thấy sau này và dựa trên phép biến đổi Fourier nhanh chóng [Guinier -1991]).

Phương pháp Lucas - Lehmer (1930) bao gồm các nội dung sau:

Trình tự lặp lại được xác định:

$$E_1 = 4$$

$$E_{n+1} = (E_n)^2 - 2$$

Một số thành viên đầu tiên của loạt bài này là:

$$4, 14, 194, 37634, \dots$$

Định lý 1.7 (Lucas-Lemmer). *Số tự nhiên $m = 2^p - 1$ là số nguyên tố Mersenne (với p lẻ) nếu và chỉ khi:*

$$(E_{p-1})\% (2^p - 1) = 0$$

Chúng ta cho phép người đọc triển khai một chương trình để tìm một vài số nguyên tố Mersenne đầu tiên. Trong đoạn tiếp theo, chúng ta sẽ xem xét một trong những ứng dụng của số nguyên tố Mersenne - để tìm kiếm các số hoàn hảo.

Bài tập

- ▷ 1.30. Viết chương trình tìm n số Mersenne đầu tiên mà không sử dụng định lý Lucas-Lemmer.
- ▷ 1.31. Viết chương trình tìm n số Mersenne đầu tiên sử dụng định lý Lucas-Lemmer. Để so sánh hiệu quả với bài toán trước đó.
- ▷ 1.32. $2^n - 1$ có thể là số nguyên tố nếu n không phải là số nguyên tố không?

1.3.2. Số hoàn thiện

Định nghĩa 1.15. Một số tự nhiên n được gọi là *hoàn hảo* nếu nó bằng tổng của tất cả các ước của nó (bản thân n không được coi là ước).

3 số hoàn hảo đầu tiên là:

$$6 = 1 + 2 + 3,$$

$$28 = 1 + 2 + 4 + 7 + 14,$$

$$496 = 1 + 2 + 4 + 8 + 16 + 31 + 62 + 124 + 248.$$

Số lượng các số hoàn hảo tăng rất nhanh:

$$8128, 33550336, 8589869056, \dots$$

Vì vậy, nếu chúng ta cố gắng tìm kiếm chúng bằng một thuật toán trực tiếp (liên tiếp với từng $n = 1, 2, 3, \dots$ để kiểm tra xem nó có hoàn hảo hay không), chúng ta sẽ không thể tìm được nhiều hơn số hoàn hảo thứ 5 cho hợp lý thời gian tính toán. Sau đây là sự hỗ trợ của chúng ta:

Định lý 1.8 (Euler). *Nếu $2^p - 1$ là số nguyên tố thì $2^{p-1}(2^p - 1)$ là hoàn hảo.*

Theo định lý trên, một chương trình tìm n số hoàn hảo đầu tiên có thể được biên soạn trực tiếp: chỉ cần biết lũy thừa p_i của n số Mersenne đầu tiên (10 lũy thừa đầu tiên là $2, 3, 5, 7, 13, 17, 19, 31, 61, 89$). Vấn đề này sinh bởi vì ngay cả ở các giá trị nhỏ của n , các số hoàn hảo được yêu cầu vượt quá giá trị lớn nhất cho phép đối với một kiểu số nguyên trong ngôn ngữ C. Do đó, chúng ta sẽ không sử dụng các kiểu tiêu chuẩn, nhưng sẽ thực hiện các phép toán cần thiết với các số "lớn": nhân một số với 2 và tăng một. Chúng ta sẽ giữ số "lớn" trong mảng một chiều number[]: mỗi phần tử của nó đại diện cho một chữ số thập phân. Để thuận tiện, các chữ số của số được sắp xếp theo thứ tự ngược lại trong mảng: chữ số đầu tiên được viết trong number[k-1] (nếu số có k chữ số), chữ số thứ hai trong number[k-2], v.v. Chữ số k cuối cùng được viết bằng number[0]. Thuật toán để tăng một số được viết bằng number[] như sau:

Tăng một đơn vị

```
i = 0; number[i]++;
while(10==number[i])(number[i]=0; number[+i]++;
if (i == k) k++;
```

Vì trong tìm kiếm các số hoàn hảo chỉ thực hiện phép nhân với 2 nên kết quả sẽ là lũy thừa của cặp số, tức là chữ số cuối cùng không được là 9, thì hai dòng cuối cùng, bắt đầu từ dấu kiểm **while** (`10 == number[1]`), là thừa.

Tương tự, chúng ta nhân một số lớn với hai:

Nhân một số lớn với hai

```
unsigned i, carry = 0, temp;
for (i = 0; i < k; i++) {
    temp = number[i] * 2 + carry;
    number[i] = temp % 10;
    carry = temp / 10;
}
if (carry > 0) number[k++] = carry;
```

Sau đây là một triển khai hoàn chỉnh tìm 10 số hoàn hảo đầu tiên, với lũy thừa của mươi số nguyên tố Mersenne đầu tiên được đặt làm hằng số trong mPrimes[]

Chương trình 1.14. Số hoàn thiện (117perfect.c)

```
#include <stdio.h>
#define MN 10
unsigned mPrimes[MN]={2,3,5,7,13,17,19,31,61,89 };
unsigned k, number[200];

void doubleN(void)
{ unsigned i, carry = 0, temp;
  for (i = 0; i < k; i++) {
      temp = number[i] * 2 + carry;
      number[i] = temp % 10;
      carry = temp / 10;
  }
  if (carry > 0) number[k++] = carry;
}

void print(void)
{ unsigned i;
  for (i = k; i > 0; i--) printf("%u", number[i-1]);
  printf("\n");
}
```

```

void perfect(unsigned s, unsigned m)
{
    unsigned i;
    k = 1; number[0] = 1;
    for (i = 0; i < m; i++) doubleN(); //đây là ước số có dạng $2^i $ 
    number[0]--;/những chữ số cuối cùng chắc chắn giữa $ \{2,4,8,6\} $ 
    for (i = 0; i < m - 1; i++) doubleN();
    printf("So hoan hao thu %2u la = ", s);
    print(); // In ra lần lượt các số
}

int main() {
    unsigned i;
    for (i=1; i<= MN; i++) perfect(i, mPrimes[i-1]);
    return 0;
}

```

Kết quả thực hiện chương trình:

Số hoàn hảo đầu tiên là = 6

Số hoàn hảo thứ 2 là = 28

Số hoàn hảo thứ 3 là = 496

Số hoàn hảo thứ 4 là = 8128

Số hoàn hảo thứ 5 là = 33550336

Số hoàn hảo thứ 6 là = 8589869056

Số hoàn hảo thứ 7 là = 137438691328

Số hoàn hảo thứ 8 là = 2305843008139952128

Số hoàn hảo thứ 9 là = 2658455991569831744654692615953842176

Số hoàn hảo thứ 10 là = 19156194260823610729479337808430363813099732154816

Lưu ý: Rõ ràng, định lý trên chỉ cho các số hoàn hảo chẵn (thực tế là tất cả các số hoàn hảo chẵn). Câu hỏi về việc liệu có những số hoàn hảo lẻ vẫn chưa được trả lời (có lẽ là không). Người ta đã chứng minh rằng nếu có thì chúng phải có ít nhất 300 chữ số và nhiều ước.

Bài tập

- ▷ 1.33. Chứng minh rằng tổng các giá trị nghịch đảo của các ước (bao gồm cả 1 và số chính nó) của mỗi số hoàn hảo là 2. Ví dụ, với 6 ta có: $1/1 + 1/2 + 1/3 + 1/6 = 2$.

► 1.34. Viết chương trình tìm tất cả các số mà tổng giá trị nghịch đảo của các ước (kể cả 1 và chính số đó) là 2. Chúng có hoàn hảo không?

1.4. Những hệ số đa thức, tam giác Pascal và giai thừa

Định nghĩa 1.16. Cho một tập hợp n phần tử đã cho. Số của tất cả các tập con k phần tử có thể có của nó được gọi là hệ số nhị thức và được biểu thị và tính toán như sau:

$$C_n^k = C_n^k = \frac{n(n-1)\dots(n-k+1)}{k(k-1)\dots1}. \quad (1.8)$$

Chúng ta sẽ lưu ý rằng các ký hiệu C_n^k và C_n^k không hoàn toàn tương đương và ký hiệu sau tổng quát hơn ký hiệu trước, vì nó thường cho phép các giá trị thực của n , nơi không có ý nghĩa tổ hợp. Tuy nhiên, đối với mục đích của chúng ta, các ký hiệu này là tương đương và chúng ta sẽ sử dụng chúng song song. Nếu n là số tự nhiên, ta có thể viết (1.8) là:

$$C_n^k = C_n^k = \frac{n!}{k!(n-k)!}. \quad (1.9)$$

Chúng ta sẽ lưu ý rằng đây chính xác là công thức cung cấp số tổ hợp không lặp lại n phần tử của lớp k (xem 1.3.3.) Và đây không phải là ngẫu nhiên (Tại sao?).

Hệ số nhị thức có một số ứng dụng, ví dụ như trong khai triển của $(a+b)^n$, do đó tên của chúng là:

$$(a+b)^n = C_n^0 a^n b^0 + C_n^1 a^{n-1} b^1 + \dots + C_n^n a^0 b^n = \sum_{i=0}^n C_n^i a^{n-i} b^i.$$

Có một số thuộc tính và công thức liên quan đến hệ số nhị thức [Knuth-1/1968]. Chúng ta sẽ xem xét một số trong số chúng trong đoạn này.

Trong Bảng 1.3. các giá trị của C_n^k được hiển thị, với $0 \leq k \leq n < 10$.

n	C_n^0	C_n^1	C_n^2	C_n^3	C_n^4	C_n^5	C_n^6	C_n^7	C_n^8	C_n^9
0	1	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0
2	1	2	1	0	0	0	0	0	0	0
3	1	3	3	1	0	0	0	0	0	0
4	1	4	6	4	1	0	0	0	0	0
5	1	5	10	10	5	1	0	0	0	0
6	1	6	15	20	15	6	1	0	0	0
7	1	7	21	35	35	21	7	1	0	0
8	1	8	28	56	70	56	28	8	1	0
9	1	9	36	84	126	126	84	36	9	1

Bảng 1.3. Hệ số đa thức

Một số tính chất của hệ số nhị thức trở nên rõ ràng nếu chúng ta phân tích Bảng 1.3 kỹ hơn, ví dụ:

$$C_n^0 = C_n^n = 1, \quad (1.10)$$

$$C_n^k = C_n^{n-k}, \quad (1.11)$$

$$C_n^k = C_{n-1}^k + C_{n-1}^{k-1}. \quad (1.12)$$

Nếu chúng ta loại bỏ các phần tử 0 khỏi Bảng 1.3. ở trên chúng ta sẽ nhận được một tam giác được gọi là tam giác Pascal:

$n = 0$							1
$n = 1$						1	1
$n = 2$				1	2		1
$n = 3$			1	3	3	1	
$n = 4$		1	4	6	4	1	
$n = 5$	1	5	10	10	5	1	

Hình 1.8. Tam giác Pascal

Số đầu tiên và số cuối cùng trong mỗi hàng là 1 (theo sau từ thuộc tính (1.10) và (1.11)), và lấy nhau dưới dạng tổng của hai số ở

trên nó (thuộc tính (1.12)). Do đó, các hệ số của bậc n có thể được tìm thấy nếu chúng ta có bậc $(n - 1)$. Nếu chúng ta tìm $C_n^k = C_n^{k-1}$ và bắt đầu từ hàng đầu tiên, chúng ta có thể tìm thấy tất cả các hàng cho đến thứ n và giá trị ở vị trí thứ k sẽ chính xác là C_n^k . Lưu ý rằng khi tính toán các giá trị trong hàng i , chúng ta chỉ sử dụng các giá trị của hàng $i - 1$, chứ không sử dụng các giá trị trước đó.

Sau đây là cách triển khai trực tiếp trong ngôn ngữ C: chúng ta sẽ giữ hàng được tính cuối cùng trong mảng `lastLine[]`, và các tham số n và k cho C_n^k bắt buộc được định nghĩa ở đầu chương trình dưới dạng hằng số.

Chương trình 1.15. Tam giác Pascal (118pascalt.c)

```
#include <stdio.h>
// Cõ lớn nhất của tam giác
#define MAXN 1000
unsigned n = 7;
unsigned k = 3;
unsigned long lastLine[MAXN + 1];

int main(void) {
    unsigned i, j;
    lastLine[0] = 1;
    for (i = 1; i <= n; i++) {
        lastLine[i] = 1;
        for (j = i - 1; j >= 1; j--)
            lastLine[j] += lastLine[j - 1];
    }
    printf("C(%u,%u) = %lu\n", n, k, lastLine[k]);
    return 0;
}
```

Chúng ta sẽ đề xuất một thuật toán khác dựa trên công thức (1.8) và trình bày một sơ đồ hữu ích để giảm các phân số hữu tỉ có tử số và mẫu số lớn. Nếu chúng ta thử tính toán trực tiếp trên (1.9) $n!$ và $k!(n - k)!$, trong nhiều trường hợp thu được kết quả trung gian lớn, trong khi kết quả cuối cùng không phải là một số lớn như vậy: ví dụ, với $n = 100$ và $k = 2$, chúng ta sẽ phải tính $100!$ và $2!.98!$ (các số trên 150 chữ số), và kết quả của C_{100}^2 chỉ là 4950.

Thuật toán 2 để tìm C_n^k (với thừa số hóa)

1. Như một ví dụ, chúng ta sẽ xem xét C_7^3 . Chúng ta sẽ biểu diễn tử số của công thức (1.8) dưới dạng tích của các số, mỗi số chúng ta sẽ phân tích thành các thừa số nguyên tố. Bây giờ chúng ta có thể phân tích toàn bộ tác phẩm: đổi với ví dụ $7! = 1.2.3...7 = 1.2.3.2^2.5.(2.3).7 = 2^4.3^2.5^1.7^1$.
2. Tương tự, chúng ta phân tích mẫu số: $3!.(7 - 3)! = 1.2.3.1.2.3.2^2 = 2^4.3^2$.
3. Viết tắt tử số và mẫu số: $\frac{2^4.3^2.5^1.7^1}{2^4.3^2} = \frac{5^1.7^1}{1}$.
4. Sau khi rút gọn ta thực hiện các phép nhân còn lại ở tử số và được kết quả như ý: $5^1.7^1 = 35$.

Việc thực hiện thuật toán 2 như sau.

Chương trình 1.16. Tính hệ số Newton (119cnk.c)

```
#include <stdio.h>
#define MAXN 500
unsigned long n = 7;
unsigned long k = 3;
unsigned long pN, primes[MAXN], counts[MAXN];

void modify(long int x, int how)
{ unsigned i;
  for (i = 0; i < pN; i++)
    if (x == primes[i]) {counts[i] += how; return;}
  counts[pN] = how;
  primes[pN+1] = x;
}

void solve(unsigned long start, unsigned long end, unsigned long inc)
{ unsigned long prime, mul, how, i, max;
  for (i = start; i <= end; i++) {
    mul = i;
    prime = 2;
    while (mul != 1) {
      for (how = 0; mul % prime == 0; mul /= prime, how++);
      if (how > 0) modify(prime, inc * how);
      prime++;
    }
  }
}
```

```

}

long int calc(void)
{ int i, j;
  long int result = 1;
  for (i = 0; i < pN; i++)
    for (j = 0; j < counts[i]; j++) result *= primes[i];
  return result;
}

int main(void) {
  printf("C(%lu,%lu)= ", n, k);
  pN = 0;
  if (n - k < k) k = n - k;
  solve(n - k + 1, n, 1); // phân tích tử số (n-k+1),...,n
  solve(1, k, -1); // phân tích mẫu số 1,...,k
  printf("%lu\n", calc());
  return 0;
}

```

Một cách tiếp cận thuật toán như vậy (tính toán thừa số, tức là thực hiện các hành động với các số ở dạng thừa số nhân với thừa số nguyên tố) được áp dụng thành công không chỉ khi tìm C_n^k , mà luôn luôn khi thu được các giá trị trung gian lớn ở tử số và mẫu số, trong khi bản thân kết quả không phải là quá tuyệt vời.

Lưu ý: Trong chương trình trên, nếu cần, chúng ta áp dụng công thức "đảo ngược" dữ liệu đầu vào. Đồng thời, những vết cắt rõ ràng được thực hiện đầu tiên và chỉ sau đó là sự phân hủy.

Bài tập

► 1.35. Chứng minh các tính chất (1.10), (1.11) và (1.12) bằng cách sử dụng định nghĩa hệ số của nhị thức. Ví dụ, bằng chứng của (1.10) có thể được thực hiện như sau:

► 1.36. Đưa ra các chứng minh tổ hợp của các tính chất (1.10), (1.11) và (1.12).

► 1.37. Thuật toán đề xuất 1 (pascalt.c) yêu cầu một mảng lastLine[] với $n + 1$ phần tử để tìm C_n^k và ở các giá trị cao hơn của n , việc cấp

phát nhiều bộ nhớ như vậy sẽ không thể thực hiện được. Có thể sửa đổi nó theo cách sau: vòng lặp bên trong, lặp đầy hàng tiếp theo của tam giác, nên được thực hiện không phải từ 1 đến i , mà từ 1 đến k , vì đoạn của tam giác không còn quan tâm đến chúng ta. Lưu ý rằng khi k gần với n , chúng ta có thể áp dụng tính chất (1.11) và tìm C_n^{n-k} thay cho C_n^k .

1.5. Hệ số đếm và sự biến đổi hệ

Nói chung, chúng ta có thể định nghĩa hệ thống số như một tập hợp các dấu hiệu đồ họa và các quy tắc để biểu diễn các số. Trong các thời đại khác nhau của sự phát triển lịch sử của nhân loại, nhiều hệ thống số khác nhau đã được sử dụng, phổ biến nhất ngày nay là hệ thống số Ả Rập. Nó sử dụng một hoặc nhiều ký hiệu đồ họa 0, 1, 2, 3, 4, 5, 6, 7, 8 và 9, được gọi là số, để đại diện cho các số tự nhiên. Thế giới Ả Rập khác với mười biểu tượng đồ họa được đề cập ở trên, nhưng nó là cùng một hệ thống số.). Số chữ số dùng để viết số sẽ được gọi là cơ sở của hệ thống. Ví dụ, hệ thống chữ số Ả Rập là hệ thập phân (sử dụng 10 chữ số). Cũng có những khái niệm khái quát về cơ số, cho phép nó là một số thực (môđun khác 0 và 1) tùy ý (hoặc thậm chí phức) với một dấu. Trong trường hợp chúng ta có $p < 0$ đối với cơ số p của hệ thống số, các số âm sẽ được viết mà không sử dụng dấu "-" một cách rõ ràng.

Các hệ thống khác đóng một vai trò đặc biệt trong số học máy tính - hệ nhị phân, hệ thập lục phân và một phần là hệ bát phân. Hệ thống số nhị phân sử dụng các chữ số 0 và 1 để viết số. Ví dụ: số thập phân $11_{(10)}$ sẽ giống như $1011_{(2)}$ (Chúng ta sẽ sử dụng dấu ngoặc đơn và một phông chữ nhỏ để chỉ ra cơ sở của hệ thống số).. Cần có 16 chữ số để viết số trong hệ thống số thập lục phân. Do đó, ngoài các chữ số từ 0 đến 9 của hệ thống số thập phân, các chữ cái in hoa A, B, C, D, E và F được sử dụng, với các giá trị 10, 11, 12, 13, 14 và 15, tương ứng. số $254_{(10)}$ được biểu thị là $FE_{(16)}$.

Bằng phương pháp phân tích toán học, có thể chỉ ra rằng hệ thống số tối ưu là hệ thống có cơ số $e = 2,718281828\dots$ (số e định nghĩa như $a_n = \left(1 + \frac{1}{n}\right)^n$ với $n \rightarrow \infty$). Ở đây, tối ưu có nghĩa là tỷ lệ tốt nhất giữa độ dài bản ghi của một số trong hệ thống tương ứng

và số chữ số được sử dụng bởi hệ thống.

Vì số e là số vô tỷ (nó không thể được biểu diễn dưới dạng tỷ số của hai số nguyên), làm việc với hệ thống này là không thuận tiện và theo nghĩa này, chúng ta có thể coi các hệ thống số tối ưu dựa trên 3 và 2. Từ quan điểm toán học, hệ thống số bậc ba là thích hợp hơn vì số 3 gần với e hơn. Tuy nhiên, từ quan điểm kỹ thuật thuận tủy, hệ thống số nhị phân được ưu tiên hơn. Hệ thống số nhị phân cực kỳ thuận tiện cho việc thực hiện kỹ thuật - ở 0 điện áp là 0V được so sánh và ở 1 – 5 V. Hầu như tất cả các máy tính hiện đại đều hoạt động trong hệ thống số nhị phân. Trước đây ở Liên Xô cũng đã có những máy móc hoạt động với hệ số đổi xứng bậc ba, ví dụ như Setun, được tạo ra vào năm 1960 tại Moscow, sử dụng các số $-1, 0$ và 1 .

Ý nghĩa của hệ thập lục phân được xác định bởi thực tế rằng 16 là lũy thừa của $2(16 = 2^4)$. Cần 4 bit (chữ số nhị phân - 0 và 1) để mã hóa các số từ 0 đến 15. Để tạo điều kiện thuận lợi cho quá trình xử lý của chúng, các bit nhị phân này được nhóm thành các byte - nhóm 8 bit. Để dàng nhận thấy rằng các số từ 0 đến 255 có thể được viết trong một byte, tức là 256 số khác nhau. Vì $256 = 16^2$ nên mỗi byte có thể được biểu diễn bằng hai chữ số thập lục phân liên tiếp. Việc chuyển từ hệ nhị phân sang hệ thập lục phân rất đơn giản - số nhị phân được chia từ phải sang trái thành các tứ phân (4 bit liên tiếp) và nếu cần thì bên trái được bổ sung bằng các số không, sau đó mỗi tứ phân được chuyển đổi riêng rẽ thành một hệ thập lục phân tương ứng chữ số.

Chúng ta sẽ minh họa phương pháp vừa được mô tả bằng cách chuyển đổi số $10101000110111_{(2)}$ sang hệ thống số thập lục phân. Chúng ta chia nó thành các số ghi chép:

$$10|1010|0011|0111$$

và thêm các số không vào nó:

$$0010|1010|0011|0111$$

Bây giờ chúng ta mã hóa mỗi số ghi chép bằng chữ số thập lục phân tương ứng của nó và thu được $2A37_{(16)}$.

Tình hình cũng tương tự khi chuyển từ hệ thống số nhị phân sang hệ bát phân, trong trường hợp đó số được chia thành các bộ ba (ba chữ số nhị phân liên tiếp). Dưới đây chúng ta sẽ xem xét một phương pháp tổng quát hơn để chuyển đổi giữa các hệ thống số.

Lưu ý rằng bản ghi số không thể xác định rõ ràng nó được viết trong hệ thống số nào. Ví dụ, đối với số 153, chúng ta chỉ có thể nói rằng nó có ít nhất là 6. Thực tế, dựa trên ý nghĩa của các số 1, 5 và 3, chúng ta có thể nói rằng số đó được viết trong một hệ thống dựa trên ít nhất 3, bởi vì Có 3 chữ số khác nhau, có thể được hiểu là 0, 1 và 2. Để tránh những hiểu lầm như vậy, theo thông lệ, chúng ta chỉ định trong ngoặc sau số đó bằng phông chữ nhỏ hơn, như trên, cơ sở của hệ thống được sử dụng.

Hệ thống chữ số Ả Rập là vị trí, tức là giá trị của các chữ số không được xác định chặt chẽ và thay đổi tùy thuộc vào vị trí của chữ số. Ví dụ, trong số 123, số 3 có giá trị là 3, trong khi ở số 34, nó có giá trị là 30. Cũng có các hệ thống số không vị trí (xem 1.1.7.). Trong đó giá trị của mỗi dấu hiệu được cố định nghiêm ngặt và không phụ thuộc vào vị trí mu.

Nói chung có hai phương pháp để viết một số trong một hệ thống số: *kỹ thuật số* và *đa thức*. Ghi âm kỹ thuật số thường được ưu tiên là ngắn hơn. Trong trường hợp này, các chữ số tương ứng của số được viết liền kề nhau và giá trị của chúng tăng từ phải sang trái theo một cấp số nhân hình học với một phần p, trong đó p là cơ sở của hệ thống số được sử dụng. Ký hiệu số có dạng $a_n a_{n-1} \dots a_0(p)$, trong đó $a_i (1 \leq i \leq n)$ là một chữ số. Trong một bản ghi đa thức, số có dạng:

$$A = a_n p^n + a_{n-1} p^{n-1} + \dots + a_1 p + a_0$$

Kí hiệu đa thức rất hữu ích khi chuyển từ hệ thống số này sang hệ thống số khác, như chúng ta sẽ thấy bên dưới.

Bài tập

- ▷ 1.38. Tại sao cơ sở của hệ thống số phải là môđun khác 0 và 1?
- ▷ 1.39. Để trình bày các số 17 và -17 trong một hệ thống số, hãy dựa vào: 2; 8; 16.
- ▷ 1.40. Để trình bày các số 17 và -17 trong một hệ thống số, dựa

vào $-2; -8; -16$.

- ▷ **1.41.** Không cần thông qua hệ thập phân, chuyển đổi các số nhị phân: 111, 110100, 1110100101, 10010101, 10101010101 và 1011110101 sang hệ thống số thập lục phân.
- ▷ **1.42.** Không cần thông qua hệ thập phân, chuyển các số nhị phân: 11, 11001, 1010101, 111111, 1010101000, 10101101000 và 11010111000 thành một hệ thống số bát phân.
- ▷ **1.43.** Kí hiệu đa thức của một số trong hệ thống số đôi xứng có dạng như thế nào?
- ▷ **1.44.** Biểu diễn các số 17 và -17 trong một hệ số đôi xứng bậc ba.

1.5.1. Chuyển hệ cơ số 10 sang cơ số p

Dựa vào cách biểu diễn một số ở dạng đa thức, ta có thể xây dựng thuật toán chuyển từ hệ thập phân sang hệ số p như sau: chia số A cho p theo thương và dư cho đến khi A trở thành 0, sau đó viết ngược lại số dư. đơn đặt hàng. biên nhận của họ (Tại sao?). Ví dụ, khi chuyển đổi số 29 thành một hệ thống số nhị phân, chúng ta nhận được (với chữ in nghiêng dưới mỗi phép chia là phần dư tương ứng):

$$29 : 2 = 14 : 2 = 7 : 2 = 3 : 2 = 1 : 2 = 0$$

$$\begin{array}{cccccc} 1 & 0 & 1 & 1 & 1 \end{array}$$

Ta lấy các số dư còn lại thu được theo thứ tự ngược lại ta được: $29_{(10)} = 11101_{(2)}$.

Hàm convert () thực hiện chuyển đổi và trả về kết quả là một chuỗi ký tự:

Hàm chuyển đổi convert() trong 120base.c

```
char getChar(char n) //Trả về ký hiệu tương ứng với $n$  
{ return (n < 10) ? n + '0' : n + 'A' - 10; }  
  
void reverse(char *pch)  
{ char *pEnd;
```

```

for (pEnd = pch + strlen(pch) - 1; pch < pEnd; pch++, pEnd--) {
    char c = *pch;
    *pch = *pEnd;
    *pEnd = c;
}
}

void convert(char *rslt, unsigned long n, unsigned char base)
//Chuyển đổi số nguyên hệ thập phân n (n >= 0)
//thành hệ đếm có cơ sở
{ char *saveRslt = rslt ;
while (n > 0) {
    *rslt ++ = getChar((char)(n % base));
    n /= base;
}
*rslt = '\0';
reverse(saveRslt);
}

```

Nếu số nhỏ hơn 1, lại có một ký hiệu ở dạng đa thức, trong trường hợp đó sẽ thu được một đa thức *bậc âm* của *p*. Khi chuyển *A* sang hệ số *p*, mọi thứ hoàn toàn ngược lại với trường hợp của một số tự nhiên: Ta nhân *A* với *p*, tách các phần nguyên và viết chúng theo thứ tự nhận. Ví dụ: chuyển đổi 0,125 thành một hệ thống số nhị phân trông giống như sau:

$$0,125_2 = 0,25 \cdot 2 = 0,5 \cdot 2 = 1,0$$

Ta được $0,125_{(10)} = 0,001_{(2)}$. Ở đây một vấn đề khác nảy sinh - ký hiệu của *A* trong hệ số thứ *p* có thể là một phân số vô hạn (không tuần hoàn). Do đó, sẽ là khôn ngoan nếu giới thiệu một số chính xác số chữ số tối đa sau dấu thập phân. Hàm convertLessThan1() chuyển đổi số *A* ($0 \leq A < 1$) thành cnt chữ số gần nhất sau dấu thập phân, bỏ qua các chữ số sau.

Hàm chuyển đổi convertLessThan1() trong 120base.c

```

void convertLessThan1(char *rslt,
double n,
unsigned char base,
unsigned char cnt)
//Chuyển một số  $0 \leq n < 1$  vào hệ đếm với cơ số

```

```
//không lớn hơn cnt của số chữ số sau dấu phẩy thập phân.
{
    while (cnt--) {
        //Phải chăng ta không nhân 0?
        if (fabs(n) < EPS) break;
        //Nhận chữ số tiếp theo
        n *= base;
        * rslt ++ = getChar((char)(int)floor(n));
        n -= floor(n);
    }
    * rslt = '\0';
}
```

Sau khi thực hiện các hàm chuyển đổi riêng biệt cho các trường hợp $A > 1$ và $0 \leq A < 1$, nó vẫn tổ hợp chúng theo cách tự nhiên, cho phép chuyển đổi bất kỳ số thực nào, kể cả các số âm. Chúng ta nhận được hàm convertReal()

Hàm chuyển đổi convertReal() trong 120base.c

```
void convertReal(char *rslt, double n,unsigned char base,unsigned
char cnt)
//Biến đổi một số thực n thành hệ đếm cơ số cơ bản
{ double integer, fraction;
// Tìm dấu của số
if (n < 0) {
    * rslt ++ = '-';
    n = -n;
}
//chia phần nguyên và phần thập phân
fraction = modf(n, &integer);
// Chuyển đổi phần nguyên
convert(rslt, (unsigned long)integer, base);
// Đặt dấu chấm thập phân theo dấu phẩy
if ('\0' == * rslt) * rslt ++ = '0';
else rslt += strlen(rslt);
* rslt ++ = '.';
// Chuyển phần thập phân
convertLessThan1(rslt, fraction, base, cnt);
if ('\0' == * rslt) {
    * rslt ++ = '0';
}
```

```

    * rslt = '\0';
}
}

```

Bài tập

- ▷ **1.45.** Tìm kí hiệu thập phân của số $126_{(8)}; 10101_{(2)}$; $3F2B_{(16)}; 3CB_{(14)}$.
- ▷ **1.46.** Tìm kí hiệu thập phân của số $0,233_{(8)}; 0,01_{(2)}$; $0,34_{(16)}; 0,2A_{(14)}$.
- ▷ **1.47.** Tìm kí hiệu thập phân của số $126,233_{(8)}; 10101.01_{(2)}$; $3F2B,34_{(16)}; 3CB,2A_{(14)}$.
- ▷ **1.48.** Để chứng minh thuật toán được đề xuất để chuyển đổi từ p -cơ số sang thập phân.

1.5.2. Chuyển hệ cơ số p vào cơ số 10. Sơ đồ Horner

Việc chuyển đổi từ chữ số p sang hệ thống số thập phân được rút gọn để tính giá trị của đa thức từ ký hiệu đa thức của A , vì tất cả các phép toán số học được thực hiện trong hệ thống số thập phân. Ví dụ, đối với $12734_{(8)}$, chúng ta nhận được như sau:

$$12734_{(8)} = 1 \cdot 8^4 + 2 \cdot 8^3 + 7 \cdot 8^2 + 3 \cdot 8 + 4 = 5596_{(10)}$$

Lưu ý rằng chúng ta cần 10 phép nhân để tính toán. Chúng ta có thể giảm đáng kể con số này nếu chúng ta giữ các lũy thừa đã được tính ở mức 8, thay vì đếm lại chúng mỗi lần. Tuy nhiên, có một phương pháp khác tổng quát hơn cho phép chúng ta tính giá trị của đa thức bậc n với không quá n phép nhân n công thức của Horner. Ý tưởng là sử dụng (2) thay vì loại (1) trong phép tính.

$$(1) P_n(x) = a_0x^n + a_1x^{n-1} + \cdots + a_{n-1}x + a_n$$

(2) $P_n(x) = a_n + x(a_{n-1} + x(a_{n-2} + \cdots + x(a_2 + x(a_1 + xa_0))\ldots))$
- công thức của Horner

Một ví dụ về triển khai cách tiếp cận này là hàm calculate (1):

Hàm tính toán calculate() trong 120base.c

```
char getValue(char c) //Trả về giá trị của ký hiệu
{ return (c >= '0' && c <= '9') ? c - '0' : c - 'A' + 10; }
unsigned long calculate(const char *numb, unsigned char base)
//Tìm giá trị thập phân của số numb, được cho trong hệ đếm với cơ số numb>=0
{ unsigned long result;
  for (result = 0; '\0' != *numb; numb++)
    result = result*base + getValue(*numb);
  return result;
}
```

Theo cách tương tự, chúng ta có thể nhận ra phép biến đổi một số không âm nhỏ hơn 1. Trường hợp này không khác nhiều so với trường hợp biến đổi một số tự nhiên. Sự khác biệt duy nhất là ở đây đa thức có bậc âm của p .

Hàm chuyển đổi calculateLessThan1() trong 120base.c cho số âm

```
double calculateLessThan1(const char *numb, unsigned char base)
//Tìm giá trị thập phân của số numb (0<numb<1), cho trong hệ số đếm với cơ số
{ const char *end;
  double result;
  for (end=numb+strlen(numb)-1, result=0.0; end>=numb; end--)
    result = (result + getValue(*end)) / base;
  return result;
}
```

Cuối cùng, vẫn tổ hợp hai trường hợp để có được một hàm chuyển đổi một số thực tùy ý với một dấu:

Hàm tổ hợp calculateReal() trong 120base.c cho số âm

```
double calculateReal(char *numb, unsigned char base)
//Tìm giá trị thập phân của số thực numb, cho trong hệ đếm với cơ số
{ char *pointPos;
  char minus;
  double result;
  //Kiểm tra dấu âm
  if ('-' == *numb)
```

```

minus = -1;
numb++;
}
else
minus = 1;
if (NULL == (pointPos = strchr(numb, '.')))
    return calculate(numb, base); //Không có phần thập phân
//Tính phần nguyên
*pointPos = '\0';
result = calculate(numb, base);
*pointPos = '.';
//thêm vào phần thập phân
result += calculateLessThan1(pointPos+1, base);
return minus * result;
}

```

Bài tập

- ▷ **1.49.** Tìm kí hiệu thập phân của số $126_{(8)}$; $10101_{(2)}$; $3F2B_{(16)}$; $3CB_{(14)}$.
- ▷ **1.50.** Tìm kí hiệu thập phân của số $0,233_{(8)}$; $0,01_{(2)}$; $0,34_{(16)}$; $0,2A_{(14)}$.
- ▷ **1.51.** Tìm kí hiệu thập phân của số $126,233_{(8)}$; $10101.01_{(2)}$; $3F2B,34_{(16)}$; $3CB,2A_{(14)}$.
- ▷ **1.52.** Để chứng minh thuật toán được đề xuất để chuyển đổi từ hệ thống số p sang số thập phân.

1.6. Chữ số la mã

Nhiều hệ thống được thảo luận ở trên, mặc dù khác nhau về cơ sở của chúng, đều thuộc cùng một lớp: vị trí. Đôi với họ, cùng một hình có các giá trị khác nhau tùy thuộc vào vị trí của nó. Tuy nhiên, có những hệ thống số khác, được gọi là không có vị trí, trong đó giá trị của các chữ số riêng lẻ là cố định và không phụ thuộc vào vị trí của nó. Một ví dụ kinh điển về vấn đề này là hệ thống chữ số La Mã. Nó sử dụng 7 chữ cái Latinh viết hoa để biểu thị các số tự nhiên

trong phạm vi [1; 3999] (các giá trị tương ứng của chúng được cho trong ngoặc): I (1), V (5), X (10), L (50), C (100), D (500) và M (1000). Ví dụ, số 1989 được ghi là MCMLXXXIX.

1.6.1. Biểu diễn số thập phân thành chữ số La mã

Chúng ta sẽ không đi sâu chi tiết vào các thuộc tính của hệ thống số La Mã. Chúng ta sẽ chỉ lưu ý rằng khi một hình nhỏ hơn đứng trước một hình lớn hơn, giá trị của nó sẽ bị trừ đi giá trị của hình lớn hơn. Nếu không, hãy thêm:

$$XI = 10 + 1 = 11$$

$$IX = 10 - 1 = 9$$

Mỗi chữ số có thể được sau bởi bất kỳ chữ số nào nhỏ hơn. Tuy nhiên, có những giới hạn đối với số lượng chữ số liên tiếp của cùng một giá trị, cũng như chữ số nhỏ hơn có thể đứng trước chữ số lớn hơn nào. Điều này được thấy rõ trong Bảng 1.4.

1(I)	2(II)	3(III)	4(IV)	5(V)	6(VI)	7(VII)	8(VIII)	9(IX)
10(X)	20(XX)	30(XXX)	40(XL)	50(L)	60(LX)	70(LXX)	80(LXXX)	90(XC)
100(C)	200(CC)	300(CCC)	400(CD)	500(D)	600(DC)	700(DCC)	800(DCCC)	900(CM)

Bảng 1.4. Ghi lại một số số bằng chữ số La mã.

Chương trình 1.17. Chuyển số thập phân thành La Mã và ngược lại (121rom2dec.c)

```
#define MAX_ROMAN_LEN 20
#include <stdio.h>
#include <string.h>
const char *roman1_9[] = {"", "A", "AA", "AAA", "AB", "B", "BA", "BAA", "BAAA", "AC"};
const char *romanDigits[] = {"IVX", "XLC", "CDM", "M"};
const char *roman2test = "MCMLXXXIX";
void getRomanDigit(char *rslt, char x, unsigned char power)
{
    const char *pch;
    for (pch = roman1_9[x]; '\0' != *pch; pch++)
        *rslt++ = romanDigits[power][*pch - 'A'];
    *rslt = '\0';
}
```

```

//Chuyển số cơ số 10 thành số La Mã
char *decimal2Roman(char *rslt, unsigned x)
{ unsigned char power;
  char buf[10];
  char oldRslt[MAX_ROMAN_LEN];
  for (*rslt = '\0', power = 0; x > 0; power++, x /= 10) {
    getRomanDigit(buf, (char)(x % 10), power);
    strcpy(oldRslt, rsbt );
    strcpy(rsbt, buf);
    strcat(rsbt, oldRslt);
  }
  return rsbt;
}
//Chuyển số La Mã thành số cơ số 10
unsigned roman2Decimal(const char *roman, char *error)
{ unsigned rsbt, value, old;
  const char *saveRoman = roman;
  char buf[MAX_ROMAN_LEN];
  old = 1000; rsbt = 0;
  while ('\0' != *roman) {
    switch (*roman++) {
      case 'I': value = 1; break;
      case 'V': value = 5; break;
      case 'X': value = 10; break;
      case 'L': value = 50; break;
      case 'C': value = 100; break;
      case 'D': value = 500; break;
      case 'M': value = 1000; break;
      default:
        *error = 1;
        return (unsigned)(-1);
    }
    rsbt += value;
    if (value > old)
      rsbt -= 2*old;
    old = value;
  }
  return (*error = strcmp(saveRoman, decimal2Roman(buf, rsbt)))
    ? (unsigned)(-1) : rsbt;
}

```

```

int main(void) {
    unsigned decimal;
    char error;
    decimal = roman2Decimal(roman2test,&error);
    if (error) printf("chuong trinh bi loi!");
    else printf("Ket qua chuyen doi %u", decimal);
    return 0;
}

```

Bài tập

- ▷ 1.53. Viết bằng số La mã: 10; 19; 159; 763; 1991; 1979; 1997; 2002
- ▷ 1.54. Viết dưới dạng số La mã: 0; -10; 0,28; 3,14; 1/7.
- ▷ 1.55. Chứng minh thuật toán được đề xuất để chuyển đổi một số thập phân thành một hệ thống chữ số La Mã.

1.6.2. Chuyển đổi chữ số La Mã sang số thập phân

Ở đây mọi thứ đơn giản hơn. Lần này chúng ta sẽ xem xét các con số từ trái sang phải, chúng ta sẽ tính toán giá trị thập phân của chúng, và sau đó chúng ta sẽ tích lũy nó một số tiền. Nếu nó chỉ ra rằng chữ số trước đó có giá trị thập phân thấp hơn giá trị hiện tại, điều này có nghĩa là chữ số trước đó đáng lẽ không được thêm vào mà phải bị trừ đi. Vì trong trường hợp này, chúng ta đã cộng nó một lần, chúng ta nên trừ nó đi hai lần:

if (value > old) rslt -- 2 * old;

Ví dụ, đối với số 19 (XIX), con số I phải được trừ đi tổng, trong khi đối với số 21 (XXI) - nó phải được thêm vào.

Một vấn đề khác nảy sinh: làm thế nào để kiểm tra xem dãy số La Mã được cho dưới dạng tham số của roman2Decimal() có phải là một số La Mã chính xác hay không? Việc kiểm tra này rất phức tạp, nhưng chúng ta có thể làm cho mọi thứ đơn giản hơn nhiều. Với mục đích này, chỉ cần thực hiện phép biến đổi nghịch đảo và so sánh kết quả thu được với số ban đầu là đủ. Chi tiết có thể được nhìn thấy từ việc thực hiện được đề xuất.

Hãy xem nội dung hàm số

`unsigned roman2Decimal(const char *roman, char *error)`
trong chương trình trên.

Bài tập

- ▷ 1.56. Viết các chữ số La Mã trong hệ thập phân: DCLXXXIV, DC-CLXIV, LX, LXX, LXXX, XL, XXL, XXXL.
- ▷ 1.57. Để chứng minh thuật toán được đề xuất để chuyển đổi một số La Mã thành một hệ thống số thập phân.
- ▷ 1.58. Viết chương trình kiểm tra dãy số La Mã có phải là chữ số La Mã viết đúng mà không qua hệ thống số thập phân hay không. Để làm được điều này, hãy quan sát kỹ các mẫu trong Bảng 1.4.

1.7. Hồi quy và lặp lại

Một câu nổi tiếng trong lập trình dân gian là: "Để xác định khái niệm đệ quy, trước tiên chúng ta phải xác định khái niệm đệ quy." Có rất nhiều "định nghĩa" như vậy trong thế giới UNIX (ví dụ, GNU là viết tắt của GNU không là UNIX, WINE là viết tắt của WINE không là Emulator, v.v.).

Định nghĩa 1.17. Một đối tượng được gọi là *đệ quy* nếu nó được chứa trong chính nó hoặc được định nghĩa bởi chính nó.

Trong tin học máy tính, đệ quy là một trong những kỹ thuật lập trình mạnh mẽ nhất: nó định nghĩa các thuật toán một cách trang nhã, tạo ra các cấu trúc dữ liệu thuận tiện và linh hoạt.

Trong nhiều trường hợp, việc sử dụng đệ quy có thể dẫn đến các thuật toán không hiệu quả. Mục đích của đoạn này, ngoài việc giới thiệu đệ quy như một cách tiếp cận thuật toán cơ bản, sẽ là khám phá câu hỏi khi nào ứng dụng của nó hữu ích trong thực tế.

Phương tiện của biểu thức đệ quy trong chương trình C là các hàm. Nếu trong phần thân của một hàm P đã cho có tham chiếu đến chính nó, chúng ta nói rằng nó trực tiếp đệ quy. Một kịch bản "phức tạp hơn" cũng có thể xảy ra: hàm P_1 chuyển thành hàm P_2 ,

P_2 thành P_3, \dots, P_n thành P_1 . Trong trường hợp này chúng ta nói rằng P_1 , cũng như P_2, P_3, \dots, P_n , là các hàm đệ quy gián tiếp (gián tiếp).

Khi lập trình các hàm đệ quy, một số điều kiện quan trọng phải được xem xét:

1. bài toán chúng ta đang xem xét nên được chia thành các bài toán con mà (đệ quy) cùng một thuật toán có thể được áp dụng. Kết hợp các giải pháp của tất cả các bài toán phụ sẽ dẫn đến một giải pháp của bài toán ban đầu.
2. Thực hiện một thuật toán đệ quy, chúng ta phải chắc chắn rằng sau một số bước hữu hạn, chúng ta sẽ đạt được một kết quả cụ thể, tức là phải có một số hữu hạn các trường hợp đơn giản (ít nhất một) mà lời giải của chúng có thể được tìm thấy trực tiếp. Thông thường người ta gọi chúng là đáy của đệ quy.
3. Tất cả các bài toán phụ của bài toán phải "kham khát" một trong những trường hợp đơn giản này, tức là sau một số lần gọi đệ quy hữu hạn để đạt đến đáy của đệ quy (Tương tự, trong các công thức truy hồi, cũng như trong suy luận quy nạp, cần phải có một cơ sở.).

Một vài ví dụ đầu tiên về các hàm C đệ quy mà chúng ta sẽ trình bày sẽ chứng minh việc tính toán các hàm toán học được xác định tuần hoàn.

1.7.1. Tính giai thừa

Định nghĩa về giai thừa, cũng như phép tính lặp lại của hàm, chúng ta đã xem xét trong 1.1.1. Để thực hiện một hàm tính giai thừa đệ quy, chúng ta phải chắc chắn rằng hai điều kiện chính được đáp ứng:

- Dưới cùng của đệ quy là trường hợp đơn giản $n = 0$ - khi đó giá trị của hàm là 1.
- Trong tất cả các trường hợp khác, chúng ta giải quyết bài toán con cho $n - 1$ và nhân kết quả với n . Do đó tham số đệ quy giảm đơn điệu và sẽ đạt đến đáy của đệ quy sau một số bước hữu hạn (giữa n và 0 có một số lượng tự nhiên hữu hạn).

Sau đây là một hàm của C, tính toán đệ quy n :

Chương trình 1.18. Tính giai thừa (121factrec.c)

```
#include <stdio.h>
const unsigned n = 6;
unsigned long fact(unsigned i)
{ if (i < 2) return 1;
return i * fact(i - 1);
}
int main(void)
{
printf("%u! = %lu \n", n, fact(n));
return 0;
}
```

Theo cách tương tự, chúng ta có thể tính toán một cách đệ quy tổng của n số tự nhiên đầu tiên (trực tiếp bằng định nghĩa truy hồi từ 1.1.1.):

Tính tổng đệ quy

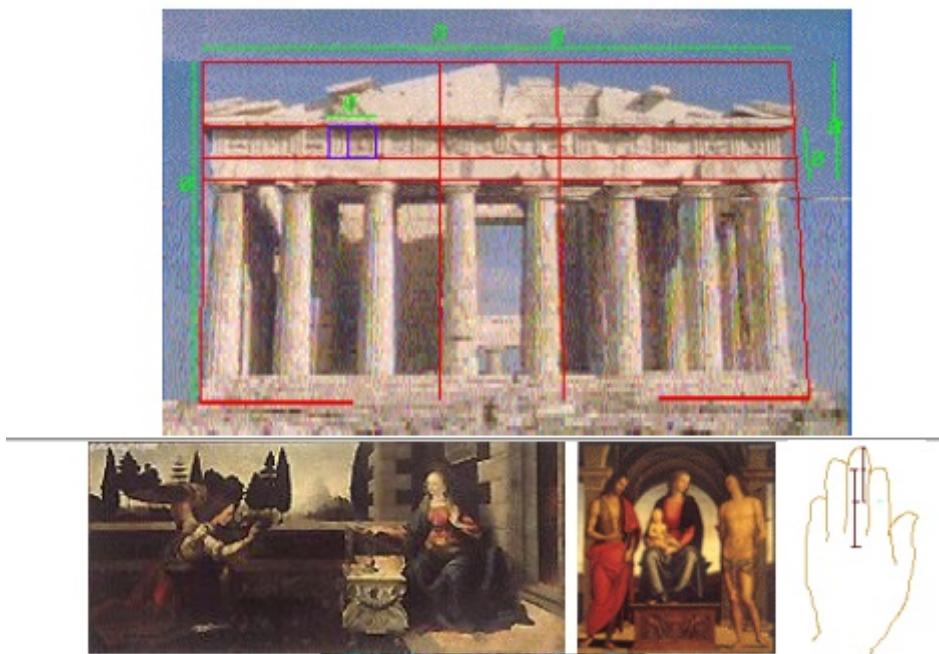
```
unsigned long sum(unsigned n){
if (0 == n) return 0;
else return n + sum(n - 1);
}
```

Có thể nhận thấy một số điều trong các triển khai trên: khi một hàm toán học được xác định lặp lại được đưa ra, việc thực hiện một hàm C đệ quy tương ứng không khó, trong khi thuật toán lặp (tuần tự) để giải trong một số trường hợp, đặc biệt là khi hàm nhiều hơn phức tạp, không phải là quá rõ ràng. (xem 1.2.2.)

Mặt khác, bộ nhớ bị tiêu tốn nhiều hơn. Điều này là do mỗi lần gọi đệ quy đến hàm trong ngăn xếp hệ thống sẽ cấp phát bộ nhớ mới cho các đối số và biến cục bộ, cũng như cho các kết quả được trả về bởi hàm. Trong các ví dụ trên, n sẽ là bắt buộc. sizeof (không dấu) byte bộ nhớ cho n cuộc gọi đệ quy. Để so sánh: chương trình tìm $n!$ từ 1.1.1. đã sử dụng một biến duy nhất trong đó kết quả được tích lũy, tức là bộ nhớ kém hơn nhiều lần. Tuy nhiên, trong những ví dụ này, bộ nhớ không phải là một yếu tố quan trọng, vì sự phát triển nhanh chóng của $n!$ chúng ta sẽ nhận được tràn số nguyên dài không dấu chứ không phải là thiếu bộ nhớ.

1.7.2. Dãy Phibonacci

Số Fibonacci là một trong những ví dụ yêu thích của một số cuốn sách và sách hướng dẫn về thuật toán cho sự sang trọng của đệ quy như một kỹ thuật lập trình. Vì lý do này, đôi khi chúng ta sẽ không cưỡng lại sự cảm dỗ để đi sâu vào chi tiết hơn về lịch sử, bản chất và tính chất của chúng, mà chúng ta tin rằng sẽ không khiến người đọc khó chịu.



Hình 1.9. Số Fibonacci trong kiến trúc và nghệ thuật

Bất cứ ai đã từng tham gia vào lập trình ít nhất một chút đều không tránh khỏi việc gặp phải những con số Fibonacci. Tại sao? Có một loạt các thuật toán và bài toán lập trình, dường như không liên quan theo bất kỳ cách nào, dựa trên những con số này. Sẽ không quá lời khi nói rằng sự hiện diện và vai trò của chúng trong khoa học máy tính, toán học và tự nhiên nói chung là thực sự gây sốc. Chỉ cần biết cách quan sát, người ta có thể tìm thấy các số Fibonacci xung quanh mình: từ cách sắp xếp các đàn chim đang bay, qua các hình nón và bài thơ, bánh hương dương và các bản giao hưởng,

nghệ thuật cổ đại (xem Hình 1.9) Và máy tính hiện đại, đến tận hệ thống năng lượng mặt trời và các sàn giao dịch chứng khoán.

Chính xác thì những con số nổi tiếng này là gì? Đây là những phần tử của số sau:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \dots,$$

Mỗi phần tử của chuỗi được lấy là tổng của hai phần trước, hai phần đầu theo định nghĩa lần lượt là 0 và 1. các công thức hợp lệ:

$$F_0 = 0,$$

$$F_1 = 1,$$

$$F_i = F_{i-1} + F_{i-2}$$

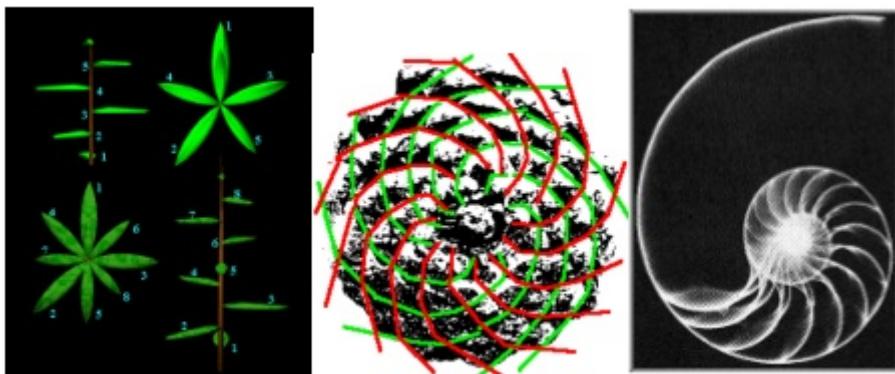


Hình 1.10. Leonardo Fibonacci.

Nhà nghiên cứu đầu tiên của dãy số trên là Leonardo Pisano (Leonardo of Pisa), hay còn được biết đến với cái tên Leonardo Fibonacci (Filius Bonacci, tức là con trai của Bonacio), người vào năm 1202 trong cuốn sách Liber Abacci (Sách đếm) đã đưa ra bài toán nổi tiếng của mình đối với thỏ. . Bài toán là tìm số thỏ sẽ có được từ một cặp trong một năm với các điều kiện sau:

- mỗi cặp thỏ đậu quả sẽ tăng thêm hai con mỗi tháng;
- thỏ mới kết trái khi được một tháng tuổi;
- Thỏ không bao giờ chết.

Như vậy, trong một tháng, chúng ta sẽ có hai cặp thỏ, sau đó là 3 cặp, tháng sau chúng ta sẽ có 5 cặp (một cặp thỏ mới ban đầu và một - của cặp đã nhận được trong tháng đầu tiên), vào tháng sau, chúng ta sẽ có tổng cộng 8 cặp đôi, v.v. Rõ ràng là quá trình được mô tả có thể diễn ra vô thời hạn. (xem Hình 1.12).



Hình 1.11. Số Fibonacci trong tự nhiên.

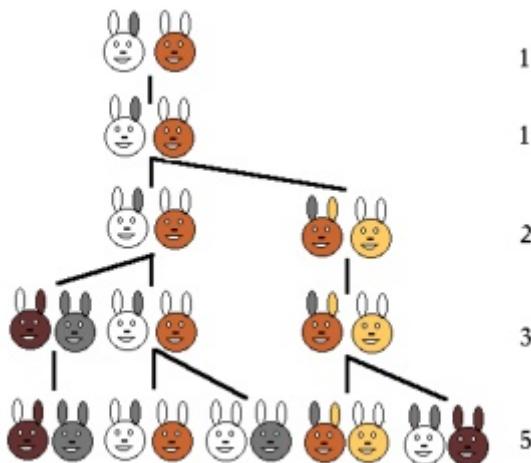
Sau đó vào năm 1611, Kepler, không biết về nghiên cứu của Fibonacci, đã khám phá lại một số nghiên cứu của mình về sự sắp xếp các lá của một số cây trên thân cây (Hình 1.11). Nói chung, như đã đề cập ở trên, số Fibonacci có bản chất cực kỳ phổ biến, rất có thể là kết quả của các quá trình tương tự như của bài toán Fibonacci đối với thỏ.

Mối quan hệ giữa số Fibonacci và thuật toán là gì? Mối liên hệ đầu tiên là chính Leonardo Fibonacci, người đã siêng năng nghiên cứu các tác phẩm của al-Khwarizmi (từ tên mà thuật toán từ bắt nguồn, như chúng ta đã đề cập trong chương giới thiệu). Sau đó, vào năm 1845, Lame sử dụng dãy Fibonacci trong quá trình nghiên cứu thuật toán Euclid nổi tiếng (xem 1.2.3) để tìm ước số chung lớn nhất. Kể từ đó, số Fibonacci dần dần bắt đầu chiếm vị trí xứng đáng trong việc phát triển và nghiên cứu các thuật toán khác nhau. Mỗi liên hệ giữa dãy Fibonacci và số sau rất thú vị

$$\phi = \frac{1 + \sqrt{5}}{2} = 1,61803$$

Số của nó có một lịch sử rất thú vị và đã được mọi người biết đến từ thời cổ đại. Euclid gọi nó là *tỷ số* của *số hữu hạn so với giá trị trung bình* và định nghĩa nó là *tỷ số* của một cặp số A và B trong đó *tỷ lệ có giá trị*:

$$\frac{A}{B} = \frac{A+B}{A} (= \phi)$$

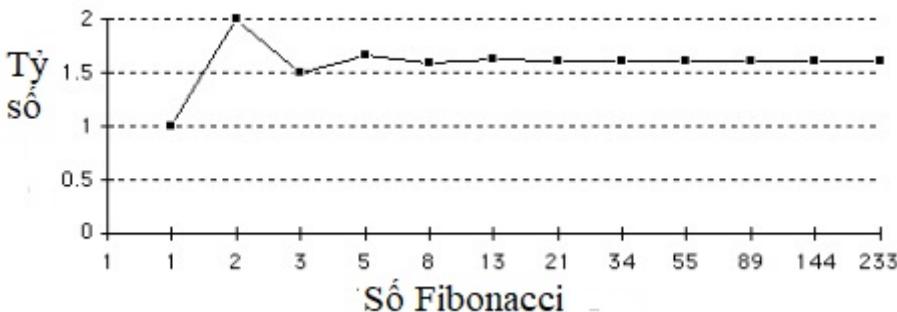


Hình 1.12. Bài toán những con thỏ.

Sau đó, vào thời kỳ Phục hưng, số ϕ được coi là một tỷ lệ thần thánh, và vào thế kỷ 19, nó có tên gọi cuối cùng như ngày nay: tỷ lệ vàng. Ký hiệu được chấp nhận ϕ xuất phát từ chữ cái đầu tiên trong tên của nhà điêu khắc Hy Lạp cổ đại Phidias, người thường sử dụng tỷ lệ này trong các tác phẩm điêu khắc của mình. Không khó để nhận thấy sự tương đồng giữa các định nghĩa về số Fibonacci và tỷ lệ vàng được đề xuất ở trên. Nó chỉ ra rằng tỷ lệ của hai số Fibonacci liên tiếp có xu hướng theo tỷ lệ vàng ϕ , và điều này đôi khi được xác định là giới hạn của tỷ lệ của hai số Fibonacci liên tiếp. (xem Hình 1.12.)

Lưu ý: Trước khi chúng ta chuyển sang các thuật toán tìm số Fibonacci, cần lưu ý rằng đôi khi có một định nghĩa khác, theo đó phần tử 0 của dãy Fibonacci là 1, không phải 0. Tất nhiên, định nghĩa sau không quá quan trọng, và người đọc, nếu cần, có thể dễ dàng

thay đổi các triển khai chương trình sau theo định nghĩa được ưu tiên.



Hình 1.13. Mối quan hệ của hai số Fibonacci liên tiếp.

Có lẽ cách thực hiện tự nhiên nhất của một hàm tìm số Fibonacci thứ n có được trực tiếp từ định nghĩa lặp lại được đưa ra ở trên:

Chương trình 1.19. Tính Fibonacci (123fibrec.c)

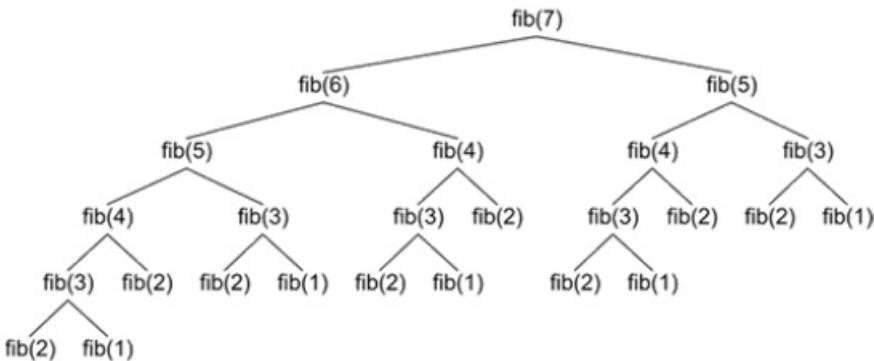
```
// da oprawq w teksta
// fib(7) dawaha razlichno - da checkna koe e po definiciqta!!!
#include <stdio.h>
const unsigned n = 7;

unsigned long fib(unsigned n)
{ if (n < 2) return 1;
  else return fib(n - 1) + fib(n - 2);
}

int main()
{
  printf("fib(%u) = %lu\n", n, fib(n));
  return 0;
}
```

Mặc dù đủ đơn giản và có vẻ tự nhiên, nhưng nhận thức như vậy là cực kỳ kém hiệu quả. Người đọc có thể thử hàm trên, ví dụ với $n = 40$. Mỗi nghịch đảo đệ quy, ngoại trừ những nghịch đảo tầm thường (đối với $n = 0$ và 1), dẫn đến hai nghịch đảo nữa. Do đó cây bài toán con phát triển theo cấp số nhân. Đồng thời, nó hoàn

toàn vò nghĩa, vì hầu hết thời gian các giá trị đã được tính toán của hàm đều được tính toán (xem Hình ??).



Hình 1.14. Cây gọi hàm đệ quy.

Số Fibonacci là một ví dụ cổ điển trong đó việc sử dụng đệ quy không thành công. Vấn đề bắt nguồn từ thực tế là các tính toán hoàn toàn không cần thiết được thực hiện, tức là nhiều thành viên của dòng được tính vài lần. Ví dụ, khi tính F_{10} , chúng ta sẽ tính F_8 hai lần - một lần khi tính $F_{10} = F_9 + F_8$ và sau đó - cho $F_9 = F_8 + F_7$. Ở các cấp độ đệ quy sâu hơn, tình hình thậm chí còn tồi tệ hơn. Ở đây, tất nhiên, đệ quy có thể cho kết quả tốt hơn đáng kể với chi phí của bộ nhớ bổ sung cần thiết. Phương pháp (được gọi là ghi nhớ) được thảo luận trong Chương 8 - Tối ưu hóa động.

Không khó để thấy rằng chúng ta có thể không sử dụng đệ quy bằng cách tính toán các số Fibonacci tuần tự: bắt đầu với số đầu tiên, chúng ta chỉ giữ lại hai phần tử được tính cuối cùng của chuỗi, vì chỉ chúng ta mới cần chúng để lấy phần tử tiếp theo. Do đó, chúng ta thu được thuật toán lặp sau:

Thuật toán lặp

```

unsigned long fibIter(unsigned n)
{ unsigned long fn = 1, fn_1 = 0, fn_2;
  while (n--) {
    fn_2 = fn_1;
    fn_1 = fn;
    fn = fn_1 + fn_2;
  }
}
  
```

```

    return fn_1;
}

```

Không khó để thấy rằng chúng ta không thực sự cần ba biến và chỉ có thể giải quyết hai biến như thế này:

Chương trình 1.20. Tính Fibonacci không đệ quy (124fibiter.c)

```

//n->const, FibIter -> fibIter
#include <stdio.h>
const unsigned n = 7;

unsigned long fibIter(unsigned n)
{ unsigned long f1 = 0, f2 = 1;
  unsigned i = 0;
  while (i < n) {
    f2 = f1 + f2;
    f1 = f2 - f1;
    i++;
  }
  return f2;
}

int main()
{
  printf("fib(%u) = %lu\n", n, fibIter(n));
  return 0;
}

```

Trong trường hợp chúng ta đang tìm n số Fibonacci đầu tiên, cách tốt nhất là sử dụng tùy chọn cuối cùng với kết quả trung gian. Nhưng chúng ta đang tìm số Fibonacci thứ n như thế nào?

Có một công thức không lặp lại cho phần tử chung của chuỗi Fibonacci, được gọi là công thức Moaver (xem 1.4.8):

$$F_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right]$$

Thoạt nhìn, công thức của Moaver có vẻ hiệu quả hơn so với phiên bản lặp lại - ở đây, thành viên thứ n của chuỗi đường như được tìm thấy trực tiếp. Thật không may, việc sử dụng nó gắn liền

với việc làm việc với các số thực, yêu cầu root và thao tác mở rộng không kém phần khó khăn, điều này làm thay đổi mọi thứ. Tuy nhiên, đối với công thức Moaver n đủ lớn, nó được ưu tiên hơn, đặc biệt nếu sử dụng phương pháp phân loại nhanh để phân loại (xem 7.3., [Nakov-1998c]). Chúng ta sẽ lưu ý rằng quá trình root của 5 có thể được thực hiện trước và kết quả có thể được đặt thành một hàng số trong chương trình. Điều này sẽ tránh tính toán nó trong quá trình thực thi, đây là một quá trình lặp lại chậm vì nó xảy ra với các hàng.

Chúng ta sẽ thảo luận về các thuật toán để tìm số Fibonacci thứ n trong Chương 8 (xem 8.1.). Ngoài việc áp dụng các kỹ thuật cụ thể cho *Tối ưu hóa động*, chúng ta sẽ xem cách chúng được kết hợp với công thức Moaver và mở rộng quy mô nhanh.

Bài tập

- ▷ **1.59.** Chứng tỏ rằng tỷ lệ của hai số Fibonacci liên tiếp có xu hướng theo tỷ lệ vàng.
- ▷ **1.60.** Để chứng minh sự tương đương của hai thực nghiệm lặp đi lặp lại của một hàm để tìm số Fibonacci thứ n: với ba và với hai biến.

1.7.3. **Ước số chung lớn nhất và thuật toán Euclid**

Ước chung lớn nhất (GCD) của hai số là một ví dụ điển hình khác về việc minh họa đệ quy như một kỹ thuật lập trình.

Định nghĩa 1.18. Hai số tự nhiên a và b đã cho. Ta nói rằng d là ước chung lớn nhất của a và b nếu nó là số tự nhiên lớn nhất chia cả a và b . Được viết chính thức, điều sau có nghĩa là đáp ứng hai điều kiện sau:

- 1) $d|a, d|b$
- 2) Nếu $d_1|a$ và $d_1|b$ thì $d_1|d$.

Trong định nghĩa, chúng ta đã sử dụng ký hiệu chia hết: $d|a$ có nghĩa là d chia một số nguyên a không có dư. Hai điều kiện của định nghĩa cũng có thể được hiểu như sau: Mỗi ước chung d_1 của a và b chia hết và d .

Chúng ta sẽ ký hiệu ước số chung lớn nhất của a và b bằng $GCD(a, b)$ hoặc đơn giản là (a, b) .

Ví dụ: $(12, 8) = (8, 12) = (8, 4) = 4, (1, 10) = (7, 10) = 1$.

Định nghĩa 1.19. Nếu GCD của hai số nguyên dương bằng 1 thì các số đó được gọi là *nguyên tố cùng nhau*.

Công thức sau đây hợp lệ cho ước chung lớn nhất của hơn hai số:

$$GCD(a_1, a_2, \dots, a_n) = GCD(a_1, a_2, \dots, a_{n-1}), a_n)$$

Chúng ta sẽ xem xét hai thuật toán để tìm $GCD(a, b)$, được đặt tên theo Euclid. Đầu tiên trong số này được gọi là thuật toán trừ Euclidean:

1) Nếu $a > b$, thực hiện 4), nếu không thực hiện 2).

2) Nếu $a = b$ thì sau đó chúng ta đã tìm được $GCD(a, b)$ - đây là giá trị của b và chúng ta kết thúc. Nếu $a \neq b$ ta thực hiện 3).

3) Gán $b = b - a$ và quay lại bước 1).

4) Gán $a = a - b$ và quay lại bước 1).

Chúng ta sẽ không đi sâu vào thuật toán này chi tiết hơn, vì thuật toán tiếp theo chúng ta sẽ xem xét là hiệu quả hơn nhiều. Nó được gọi là *thuật toán Euclid với phép chia* (nó đã được đề cập trong phần giới thiệu của cuốn sách):

Thuật toán Euclid với phép chia

Chia số nguyên a cho b được q_1 và số dư r_1 . Sau đó chia b cho phần dư thu được, rồi chia r_1 cho phần dư r_2 , v.v., cho đến khi phần dư bằng không:

$$a = q_1 \cdot b + r_1$$

$$b = q_2 \cdot r_1 + r_2$$

$$r_1 = q_3 \cdot r_2 + r_3$$

.....

$$r_{k-1} = q_{k+1} \cdot r_k + r_{k+1}, \text{ vì } r_{k+1} = 0$$

Phần dư khác không cuối cùng r_k sẽ là GCD cần thiết. Thuật toán cũng có một biến thể đệ quy tương ứng dựa trên thuộc tính

sau của GCD :

$$GCD(a, b) = GCD(b, a \% b)$$

Chương trình 1.21. Tìm ước số chung lớn nhất không đê quy (125gcditer.c)

```
#include <stdio.h>

const unsigned a = 28;
const unsigned b = 49;

unsigned gcd(unsigned a, unsigned b)
{ unsigned swap;
  while (b > 0) {
    swap = b;
    b = a % b;
    a = swap;
  }
  return a;
}

int main(void) {
  printf("NOD(%u,%u) = %u\n", a, b, gcd(a, b));
  return 0;
}
```

Chương trình 1.22. Tìm ước số chung lớn nhất đê quy (126gcdrec.c)

```
#include <stdio.h>
const unsigned a = 28;
const unsigned b = 49;

unsigned gcd(unsigned a, unsigned b)
{ return (b == 0) ? a : gcd(b, a % b);
}

int main(void) {
  printf("NOD(%u,%u) = %u\n", a, b, gcd(a, b));
  return 0;
}
```

Có thể thấy rằng hàm đệ quy ngắn và chặt chẽ hơn rất nhiều. Tuy nhiên, bộ nhớ được tiêu thụ bởi nó nhiều hơn: đối với mỗi lệnh gọi đệ quy trong ngắn xếp, bộ nhớ được cấp phát cho các tham số chính thức, cũng như cho kết quả được trả về bởi hàm.

Thuật toán Euclid nâng cao

Đôi khi, chẳng hạn như trong số học mô-đun, thuật toán Euclide được sử dụng để tìm hai số nguyên bổ sung nhân x và y ($x, y \in \mathbb{Z}$), sao cho:

$$d = GCD(a, b) = ax + by$$

Bài tập

- ▷ **1.61.**
 1. Để tìm GCD của: (10, 5); (5.10); (15, 25); (25, 15); (7, 8, 9); (3, 6, 9); (158, 128, 256); (64, 28, 72, 18).
 2. Viết chương trình rút gọn phân số thường gấp. Ví dụ, ở đầu vào $10/15$ để cung cấp cho $2/3$.
 3. Chứng minh rằng $(a_1, a_2, \dots, a_n) = (a_1, a_2, \dots, a_{n-1}), a_n$.
 4. Để thực hiện thuật toán trừ Euclide.
 5. Chứng minh rằng $(a, b) = (b, a \% b)$.
 6. Thuật toán và chương trình phải được sửa đổi như thế nào để đổi với các số nguyên a và b đã cho, ngoài GCD , tìm được hai thừa số nguyên x và y ($x, y \in \mathbb{Z}$) mà $(a, b) = ax + by$?
 7. Đề xuất và thực hiện một thuật toán khác để tìm GCD , dựa trên định lý cơ bản của số học. Đề so sánh hiệu quả của nó với hiệu quả của việc thực hiện được đề xuất ở đây tại 2; Số 5; 100; 1000 số.
 8. (Bài toán Poisson về ba bình) Cho ba bình có dung tích a, b và c , a, b và c là các số tự nhiên, $c > a > b$, $c \geq a + b - 1$, bình lớn nhất là đầy và hai cái còn lại trống rỗng. Đo d lít, với d là số tự nhiên và $0 < d \leq a$. Chỉ cho phép các tràn như vậy từ tàu này sang tàu khác, trong đó việc đổ đầy tối đa và/hoặc làm rỗng tối đa của một trong các tàu diễn ra.

1.7.4. Bội số chung nhỏ nhất

Định nghĩa 1.20. Hai số nguyên a và b đã cho. Số tự nhiên nhỏ nhất d ($d > 0$) sao cho $a|d$ và $b|d$ được gọi là bội chung nhỏ nhất LCM) của a và b .

Chúng ta sẽ đánh dấu ít nhất bội chung của a và b bằng $\text{LCM}(a, b)$ hoặc đơn giản là $[a, b]$.

Ví dụ: $[6, 15] = [15, 6] = 30, [1, 10] = 10, [5, 10] = 10, [5, 12] = 60$.

Khi chúng ta tìm LCM trên nhiều hơn hai số, có sự phụ thuộc tương tự như trong LCM:

$$\text{LCM}(a_1, a_2, \dots, a_n) = \text{LCM}(\text{LCM}(a_1, a_2, \dots, a_{n-1}), a_n)$$

Ít nhất một bội số chung có thể được tìm thấy bằng cách sử dụng mối quan hệ hiện có giữa nó và LCM, đó là:

$$\text{LCM}(a, b) = \frac{ab}{\text{LCM}(a, b)}$$

Dựa trên các tính chất trên, chúng ta sẽ thực hiện LCM của n số. Các số được cho trong mảng A[] và được cho cùng với số của chúng, dưới dạng tham số của hàm đệ quy lcm()

Chương trình 1.23. Bội chung nhỏ nhất (127lcm.c)

```
#include <stdio.h>
const unsigned n = 4;
const unsigned A[] = { 10, 8, 5, 9 };
unsigned gcd(unsigned a, unsigned b)
{
    return (0 == b) ? a : gcd(b, a % b);
}
unsigned lcm(unsigned a[], unsigned n)
{
    unsigned b;
    if (2 == n)
        return(a[0] * a[1]) / (gcd(a[0], a[1]));
    else {
        b = lcm(a, n - 1);
        return(a[n - 1] * b) / (gcd(a[n - 1], b));
    }
}

int main(void)
{
    printf("%u\n", lcm(A, n));
    return 0;
}
```

Kết quả thực hiện chương trình:

360

Bài tập

- ▷ **1.62.** Để tìm: $[10, 15]; [15, 10]; [7, 8, 9]; [3, 6, 9]; [158, 128, 256]; [64, 28, 72, 18]$.
- ▷ **1.63.** Chứng minh rằng $[a_1, a_2, \dots, a_n] = [a_1, a_2, \dots, a_{n-1}], a_n]$.
- ▷ **1.64.** Chứng minh rằng $[a, b] = \frac{ab}{(a, b)}$.
- ▷ **1.65.** Đề xuất và thực hiện một thuật toán khác để tìm NOC, dựa trên định lý cơ bản của số học. Để so sánh hiệu quả của nó với hiệu quả của việc thực hiện được đề xuất ở đây tại 2; Số 5; 100; 1000 số.

1.7.5. Trả lại giá trị từ đệ quy và dùng biến

Trong tất cả các hàm đệ quy C được trình bày cho đến nay, chúng ta chỉ thực hiện các thao tác trước lệnh gọi đệ quy. Tuy nhiên, trong thực tế, thường phải thực hiện các phép toán sau khi quay trở lại từ đệ quy.

Bài toán: In các chữ số của số tự nhiên n, đã cho trong hệ số thập phân, tuần tự từ chữ đầu tiên đến chữ số cuối cùng.

Ví dụ, với $n = 7892$, bạn sẽ cần in: 7, 8, 9, 2.

Vấn đề trong bài toán này là xác định chữ số đầu tiên của dãy số. Mặt khác, việc tìm chữ số cuối cùng và loại bỏ nó có thể chỉ với hai phép toán: $n \% 10$ và $n / 10$. Chúng ta đưa ra thuật toán sau: ở mỗi bước, chúng ta tìm chữ số cuối cùng và viết nó vào một mảng số nguyên. Sau khi viết tất cả các chữ số trong mảng, chúng ta hiển thị nó theo thứ tự ngược lại:

Chương trình 1.24. Giá trị trả về của đệ quy (128print.c)

```
#include <stdio.h>
unsigned n = 7892;
int main(void)
{
    unsigned dig[20], i, k = 0;
    while (n > 0) {
        dig[k] = n % 10;
```

```

n = n / 10;
k++;
}
for (i = k; i > 0; i--) printf("%u ", dig[i-1]);
printf("\n");
return 0;
}

```

Sử dụng đệ quy trong tác vụ này một lần nữa sẽ dẫn đến một giải pháp thanh lịch hơn nhiều. Chúng ta sẽ sử dụng thuộc tính sau: để in 7892, chúng ta cần in các chữ số của 789 và sau đó là chữ số 2. Nói chung, để in số n, chúng ta cần in (đệ quy!) Số $n/10$ và sau đó là chữ số cuối cùng $n \% 10$. Do đó, sử dụng ngăn xếp đệ quy, chúng ta không cần phải giới thiệu một mảng một cách rõ ràng. Dựa trên kết luận cuối cùng, chúng ta sẽ biên dịch hàm tương ứng của C:

Chương trình 1.25. Giá trị trả về của đệ quy (129printrec.c)

```

#include <stdio.h>
unsigned n = 7892;
void printN(unsigned n)
{
    if (n >= 10) printN(n / 10);
    printf("%u ", n % 10);
}

int main()
{
    printN(n); printf("\n");
    return 0;
}

```

Phần dưới cùng của đệ quy, khi chúng ta ngừng gọi đệ quy, là $n \leq 9$. Các phép toán biến rất quan trọng trong đệ quy. Cho đến nay, chỉ có các biến-tham số và có thể là các biến cục bộ tạm thời đã thay đổi trong phần thân của một hàm đệ quy. Các phép toán với các biến toàn cục, trước và sau khi trả về từ đệ quy, rất quan trọng trong việc thiết kế các thuật toán đệ quy. Chúng ta sẽ minh họa sự thay thế các biến đối số bằng các biến toàn cục bằng một vài ví dụ: Tìm $n!$ nó có thể được thực hiện như sau bằng cách thay thế đối số của hàm bằng một biến toàn cục i (Ưu điểm của cách triển khai đệ quy này so với những cách được thảo luận cho đến nay là không có

bộ nhớ ngăn xếp nào được sử dụng):

Chương trình 1.26. Giá trị trả về của đệ quy (122factrec.c)

```
#include <stdio.h>
const unsigned n = 6;
unsigned i;
unsigned long fact(void)
{ if (i == 1) return 1;
  return --i * fact();
}

int main(void)
{
  i = n + 1;
  printf("%u! = %lu \n", n, fact());
  return 0;
}
```

Chúng ta sẽ xem xét một ví dụ đơn giản khác: Đôi với một số tự nhiên n cho trước in ra theo thứ tự tăng dần và giảm dần các số 10^k ($1 \leq k \leq n$). Ví dụ, đối với $n = 5$, hàng sẽ được in:

10, 100, 1000, 10000, 100000, 100000, 10000, 1000, 1000, 100, 10

Chúng ta sẽ giải quyết vấn đề một cách đệ quy theo ba cách khác nhau. Với chúng, chúng ta sẽ minh họa khả năng hoán đổi cho nhau của các biến toàn cục và các tham số biến:

Phương án 1. Tất cả các biến (ngoại trừ đầu vào n) là tham số của hàm đệ quy:

Chương trình 1.27. Giá trị trả về của đệ quy (130print1.c)

```
#include <stdio.h>
const unsigned n = 5;
void printRed(unsigned k, unsigned long result)
{ printf("%lu ", result);
  if (k < n) printRed(k + 1, result * 10);
  printf("%lu ", result);
}

int main(void)
{
  printRed(1, 10);
```

```

    printf("\n");
    return 0;
}

```

Phương án 2. Tham số bộ đếm k có thể được xuất dưới dạng biến toàn cục:

Chương trình 1.28. Giá trị trả về của đệ quy (131print2.c)

```

#include <stdio.h>
const unsigned n = 5;
unsigned k = 0;
void printRed(unsigned long result)
{
    k++;
    printf("%lu ", result);
    if (k < n) printRed(result * 10);
    printf("%lu ", result);
}
int main(void) {
    printRed(10);
    printf("\n");
    return 0;
}

```

Phương án 3. Nếu chúng ta sửa đổi kết quả kết quả trước và sau cuộc gọi đệ quy, nó cũng có thể được xuất dưới dạng biến toàn cục:

Chương trình 1.29. Giá trị trả về của đệ quy (132print3.c)

```

#include <stdio.h>
const unsigned n = 5;
unsigned long result = 1;
unsigned k = 0;
void printRed(void)
{
    k++;
    result *= 10;
    printf("%lu ", result);
    if (k < n) printRed();
    printf("%lu ", result);
    result /= 10;
}
int main(void) {
    printRed();
}

```

```

    printf("\n");
    return 0;
}

```

Trong đoạn tiếp theo, khi xem xét các thuật toán tổ hợp cơ bản (cũng như nói chung trong tài liệu bên dưới), chúng ta sẽ tiếp tục sử dụng đệ quy trong tất cả các giống "kỳ lạ" của nó.

Bài tập

▷ 1.66. Để đề xuất một lý do có thể cho việc triển khai factrec.c khác nhau trong Borland C cho DOS và Microsoft Visual C ++ cho Windows. Bạn có thể đưa ra một lựa chọn "an toàn" không?

▷ 1.67. Để đưa ra giả định về kết quả của việc thực hiện đoạn phân đoạn dưới đây. Bạn có mong đợi sự khác biệt trong Borland C cho DOS và Microsoft Visual C ++ cho Windows không? Dự đoán của bạn có khớp với kết quả thực tế không?

```

unsigned i = 1;
printf("%u %u", ++i, i);

```

Và bạn nghĩ gì về mảng vỡ:

```

unsigned i = 1;
printf("%u %u", i, ++i);

```

▷ 1.68. Giả sử giá trị của biến x sau khi thực hiện đoạn chương trình dưới đây. Bạn có mong đợi sự khác biệt trong Borland C cho DOS và Microsoft Visual C ++ cho Windows không? Dự đoán của bạn có khớp với kết quả thực tế không?

```

unsigned x, a = 3, b = 5;
x = a+++b;

```

▷ 1.69. Dựa trên kết quả của các tác vụ trước đó, đưa ra các đề xuất để viết mã dễ di chuyển và rõ ràng nhất.

1.8. Thuật toán đếm cơ sở

Nhiều bài toán chúng ta sẽ xem xét trong cuốn sách này liên quan đến việc tìm ra giải pháp tối ưu (cực hạn). Trong những tình

huống như vậy, cách tiếp cận phổ biến là khám phá tất cả các giải pháp có thể chấp nhận được cho vấn đề và chọn giải pháp tối ưu - một hoặc một số. Việc tạo ra các đối tượng tổ hợp (sử dụng các thuật toán tổ hợp) tương ứng chính xác với lược đồ này.

Tuy nhiên, trong các bài toán trong đoạn này, chúng ta sẽ không quan tâm đến giải pháp cực hạn, mà chỉ quan tâm đến việc tạo ra tất cả các giải pháp: tất cả các cách có thể để sắp xếp đối tượng, tất cả các cách có thể để chọn đối tượng, v.v. Ví dụ, nếu một bộ bài 52 lá được đưa ra, chúng ta có thể quan tâm đến tất cả các cách có thể để chọn 5 lá bài sao cho chúng liên tiếp và giống nhau (theo ngôn ngữ của poker: thuần túy xô). Một ví dụ khác là tính số hoặc tạo tất cả các số điện thoại có sáu chữ số có thể có, với điều kiện bổ sung là số 9 không tham gia vào chúng.

Chúng ta sẽ xem xét các cấu hình tổ hợp nổi tiếng nhất: cả thuật toán tạo và công thức tìm số của chúng. Mặc dù thoát nhìn, các thuật toán được xem xét không trực tiếp giải quyết bất kỳ vấn đề thực tế nào ngoài các bài toán tổ hợp thuần túy, nhưng chúng (bao gồm cả kết hợp với các kỹ thuật thuật toán khác) tỏ ra cực kỳ hữu ích trong việc giải quyết một số vấn đề tối ưu hóa.

1.8.1. Hoán vị

Định nghĩa 1.21. Ta xét một tập n phần tử $A = \{a_1, a_2, \dots, a_n\}$. Mọi tập có n có thứ tự thu được với các phần tử của A , mỗi phần tử của A tham gia đúng một lần được gọi là *hoán vị*.

Tập hợp tất cả các hoán vị có thể có được ký hiệu là P_n , và số của chúng là bởi $|P_n|$. Không khó để chứng tỏ rằng $|P_n| = n!$.

Ví dụ, nếu cho trước tập hợp $\{a, b, c\}$ có 3 phần tử, thì tất cả các hoán vị có thể có (tức là tất cả n -torques có thứ tự có thể có) như sau:

$$\begin{aligned} & (a, b, c) \\ & (a, c, b) \\ & (b, a, c) \\ & (b, c, a) \\ & (c, a, b) \\ & (c, b, a) \end{aligned}$$

Đến cuối phần này, các phần tử của tập hợp chúng ta làm việc với sẽ là các số nguyên từ 1 đến n. Điều này không hạn chế tính phổ biến của các thuật toán được xem xét và chúng sẽ có giá trị đối với các tập tùy ý (bất kỳ tập n phần tử nào là đẳng cấu với tập gồm n số tự nhiên đầu tiên).

Sinh ra hoán vị

Để tạo ra các hoán vị trên máy tính, chúng ta sẽ sử dụng một thuật toán đệ quy:

Thuật toán 1:

Chúng ta đặt từng phần tử ở vị trí đầu tiên theo thứ tự tuyến tính, sau đó ở $n - 1$ vị trí còn lại, chúng ta đặt tất cả các hoán vị có thể có của $n - 1$ còn lại thành phần. Do đó, chúng ta có được sơ đồ đệ quy sau để tạo ra các hoán vị:

```
//Đặt phần tử vào vị trí i
void permute(i) {
    if (i >= n) {//Tim được một sắp xếp tuyến tính
        //những phần tử và in nó ra
        printPerm();
    } else
        for (k = 0; k < n; k++)
            if (!used[k]){//Nếu phần tử k chưa dùng đến
                used[k] = 1;//tìm được phần tử k để dùng nó
                position[i] = k; // phần tử của vị trí i là k
                permute(i+1);
                used[k] = 0; //bỏ đánh dấu phần tử k đã dùng
            }
}
```

Hàm *permute()* có một tham số duy nhất: vị trí mà chúng ta sẽ "đặt" một phần tử. Với vòng lặp bên trong, chúng ta lần lượt đặt tất cả các phần tử hiện không tham gia theo thứ tự vào vị trí thứ *i* và tiếp tục đệ quy cho vị trí thứ (*i* + 1). Chúng ta sẽ nhập một mảng *used[]* để chúng ta có thể xác định xem một phần tử có được sử dụng hay không: nếu phần tử đã tham gia vào hoán vị, thì *used[k]* == 1 và *used[k]* == 0, ngược lại (tức là . nó sẽ được "đặt"). Phần cuối của đệ quy ở *i* == *n* - khi đó tất cả các phần tử được "đặt" và chúng ta có một hoán vị, chúng ta sẽ in ra.

Đây là cách thực hiện đầy đủ, in tất cả $n!$ hoán vị trên một tập hợp có n phần tử:

Chương trình 1.30. Tính hoán vị (133permute.c)

```
#include <stdio.h>
#define MAXN 100
const unsigned n = 3;
char used[MAXN];
unsigned mp[MAXN];
void print(void)
{ unsigned i;
  for (i = 0; i < n; i++) printf("%u ", mp[i] + 1);
  printf("\n");
}
void permute(unsigned i)
{ unsigned k;
  if (i >= n) { print(); return; }
  for (k = 0; k < n; k++) {
    if (!used[k]) {
      used[k] = 1; mp[i] = k;
      permute(i+1); //Nếu có nghĩa tiếp tục sinh permute(i+1)
      used[k] = 0;
    }
  }
}
int main(void) {
  unsigned i;
  for (i = 0; i < n; i++) used[i] = 0;
  permute(0);
  return 0;
}
```

Kết quả thực hiện chương trình:

1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1

Hãy xem bình luận ở trên có chứa if. Nếu một bài toán tối ưu được đưa ra, lời giải của nó được rút gọn để tạo ra các hoán vị, có thể làm gián đoạn thế hệ hiện tại, nếu chắc chắn rằng nó sẽ không dẫn đến một lời giải tối ưu. Trong tài liệu dưới đây (xem Chương 6), chúng ta sẽ xem xét vấn đề này một cách chi tiết và tạm thời chúng ta sẽ giới hạn bản thân trong một ví dụ cụ thể về sự gián đoạn như vậy.

Bài toán: Tìm tất cả các số điện thoại có mười chữ số trong đó mỗi chữ số tham gia đúng một lần và sao cho k_i là chữ số thứ i của số điện thoại thì $\sum_{1 \leq i \leq 10} k_i = 20$.

Rõ ràng là tất cả các số điện thoại có thể được tạo ra bằng cách hoán vị. Do đó, nếu khi đặt một vài chữ số đầu tiên theo thứ tự tuyến tính, chúng ta nhận thấy rằng tổng vượt quá 20, thì sẽ không có ý nghĩa gì khi tiếp tục tạo, vì chúng ta sẽ không thể nhận được tổng chính xác.

Chương trình trên tạo ra các hoán vị theo thứ tự từ vựng. Điều này có nghĩa là cứ hai hoán vị liên tiếp (i_1, i_2, \dots, i_n) và (j_1, j_2, \dots, j_n) thì tồn tại một số k ($1 \leq k < n$) sao cho $i_p = j_p$, $p = 1, 2, \dots, k - 1$ và $i_k < j_k$. Chúng ta có thể coi hoán vị là các số trong một hệ thống số tương ứng: một hoán vị lớn hơn về mặt từ vựng so với hoán vị khác nếu nó lớn hơn dưới dạng một số.

Có thể thực hiện quá trình sinh theo cách khác: thu được các hoán vị có kích thước $k + 1$ từ các hoán vị có kích thước k , sử dụng cơ chế đệ quy. Do đó, không cần thêm bộ nhớ cho mảng used[] đã sử dụng. Điều này có thể được thực hiện, chẳng hạn, với thuật toán đệ quy sau [Nakov-1998]:

Thuật toán 2.

- 1) Tại $n = 1$, chúng ta tạo ra một hoán vị duy nhất: (1).
- 2) Xét một hoán vị (p_1, p_2, \dots, p_k) gồm k ($1 \leq k < n$) phần tử. Chúng ta đặt phần tử $(k + 1)$ ở vị trí $1, 2, \dots, (k + 1)$, tương ứng:

$$\begin{aligned}
 & (p_{k+1}, p_1, p_2, \dots, p_k) \\
 & (p_1, p_{k+1}, p_2, \dots, p_k) \\
 & \dots \\
 & (p_1, p_2, \dots, p_k, p_{k+1})
 \end{aligned}$$

Như vậy, lặp lại bước 2) với tất cả các hoán vị của k phần tử, ta thu được tất cả các hoán vị của $k + 1$ phần tử.

Việc triển khai được đề xuất dưới đây là một sửa đổi của *Thuật toán 2*. Chúng ta để nó như một bài tập để người đọc xem xét mối quan hệ giữa chúng, thoát nhìn có vẻ không trực tiếp như vậy.

Chương trình 1.31. Tính hoán vị 2 (134permswap.c)

```
#include <stdio.h>
#define MAXN 100
const unsigned n = 3;
unsigned a[MAXN];

void print(void)
{ unsigned i;
    for (i = 0; i < n; i++) printf("%u ", a[i] + 1);
    printf("\n");
}

void permut(unsigned k)
{ unsigned i, swap;
    if (k == 0) print();
    else {
        permut(k - 1);
        for (i = 0; i < k - 1; i++) {
            swap = a[i]; a[i] = a[k-1]; a[k-1] = swap;
            permut(k - 1);
            swap = a[i]; a[i] = a[k-1]; a[k-1] = swap;
        }
    }
}

int main(void) {
    unsigned i;
    printf("\n\n");
    for (i = 0; i < n; i++) a[i] = i;
    permut(n);
    return 0;
}
```

Kết quả thực hiện chương trình:

1 2 3
2 1 3
3 2 1
2 3 1
1 3 2
3 1 2

Bài tập

- ▷ **1.70.** Chứng minh rằng số hoán vị của một tập hợp n phần tử là $n!$
- ▷ **1.71.** Tìm các hoán vị của các phần tử của $\{a, b, c, d\}$ theo cách thủ công, sử dụng:
 - a) thuật toán 1
 - b) thuật toán 2
 So sánh các kết quả.
- ▷ **1.72.** Để chứng minh thuật toán 1 và thuật toán 2.
- ▷ **1.73.** Chứng tỏ rằng `permSwap.c` thực sự là một triển khai của *Thuật toán 2*.
- ▷ **1.74.** Để thực hiện một thuật toán để tạo lặp đi lặp lại các hoán vị.

1.8.2. Mã hóa và giải mã

Đôi khi, ví dụ, khi tạo các hoán vị ngẫu nhiên bằng các thuật toán heuristic (Chương 9) hoặc ghi nhớ (Chương 8), cần phải mã hóa và giải mã rõ ràng các hoán vị. Điều này có nghĩa là tại mỗi hoán vị khác nhau, một số tự nhiên duy nhất được so sánh để sau này có thể tái tạo duy nhất từ nó. Ví dụ, hãy xem xét các hoán vị có thứ tự từ vựng của ba phần tử:

$$(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1).$$

Trên mỗi chúng, chúng ta có thể so sánh một số tự nhiên duy nhất từ 0 đến 5. Nói chung, mỗi hoán vị của một tập n phần tử đã cho tương ứng với một số nhị vị giữa 0 và $n! - 1$. Một thuật toán để xác định nó bằng một hoán vị nhất định (mã hóa hoán vị) và phép

toán nghịch đảo (giải mã) có thể được biên dịch theo các sơ đồ sau [Shishkov-1995]:

Mã hóa

1) Cho `perm[]` là một mảng có độ dài n chứa các phần tử của hoán vị mà ta cần mã hóa.

2) Cho `pos = result = 0`, `p[]` là một mảng có n phần tử, $p[i] = i + 1, i = 0, 1, \dots, n - 1$.

3) Nếu $pos < n$, thì chúng ta loại trừ phần tử chiếm vị trí thứ r trong `p` và gán cho r , trong đó i là chỉ số mà `perm[pos] == p[i]`. Nếu không, thuật toán kết thúc và `result` là kết quả bắt buộc.

4) Gán `result = result * (n - pos) + r`.

5) `pos ++` và chuyển sang 3).

Giải mã

1) ho số num và chúng ta phải khôi phục lại hoán vị từ nó. Gọi `k = n-1`, `p[]` là một mảng n sao cho `p[i - 1] = i`, với $i = 1, 2, \dots, n$.

2) Trong khi $k \geq 0$, chúng ta lặp lại:

```
m = n - k;
perm [k] = num % m;
if (k > 0) num / = m;
k--;
```

3) Gán $k = 0$. Trong khi $k < n$ ta lặp lại:

```
m = perm [k];
perm [k] = p [m];
nếu (k < n)
for (i = m + 1; i < n; i++)
    p [i - 1] = p [i];
k++;
```

Sau đây là mã nguồn hoàn chỉnh của chương trình thực hiện các toán tử trên:

Chương trình 1.32. Số chữ số (135codeperm.c)

```
#include <stdio.h>
#define MAXN 100

const unsigned n = 6;
const unsigned perm[MAXN] = { 5, 3, 6, 4, 2, 1 };
```

```

const unsigned long code = 551;

unsigned long codePerm(unsigned n, unsigned perm[])
{
    unsigned p[MAXN], i, pos;
    unsigned long r, result;
    result = 0;
    for (i = 0; i < n; i++) p[i] = i + 1;
    for (pos = 0; pos < n; pos++) {
        r = 0;
        while (perm[pos] != p[r]) r++;
        result = result * (n - pos) + r;
        for (i = r + 1; i < n; i++) p[i - 1] = p[i];
    }
    return result;
}

void decodePerm(unsigned long num, unsigned n, unsigned perm[])
{
    unsigned long r, m, k;
    unsigned i, p[MAXN];
    for (i = 0; i < n; i++) p[i] = i + 1;
    k = n;
    do {
        m = n - k + 1;
        perm[k - 1] = num % m;
        if (k > 1) num /= m;
    } while (--k > 0);
    k = 0;
    do {
        m = perm[k]; perm[k] = p[m];
        if (k < n)
            for (i = m + 1; i < n; i++) p[i - 1] = p[i];
    } while (++k < n);
}

int main(void) {
    unsigned i;
    printf("Mot to hop ma hoa nhu la %lu \n", codePerm(n, perm));
    printf("Mo ma hoa cua so la %lu: ", code);
    decodePerm(code, n, perm);
    for (i = 0; i < n; i++) printf("%u ", perm[i]);
}

```

```

    printf("\n");
    return 0;
}

```

Kết quả thực hiện chương trình:

Hoán vị đã cho được mã hóa là 551

Ta giải mã hoán vị tương ứng với số 551: 5 3 6 4 2 1

Bài tập

- ▷ 1.75. 1. Sử dụng thuật toán trên, mã hóa thủ công các hoán vị: $(2, 3, 1, 4), (5, 3, 2, 4, 1), (3, 6, 4, 1, 5, 2)$.
 2. Sử dụng thuật toán trên, tìm thủ công các hoán vị của 5 phần tử tương ứng với mã: 3, 13, 27, 87, 119.

1.8.3. Hoán vị lặp lại

Bây giờ chúng ta hãy xem xét một tập đa thay vì tập A (nghĩa là có thể cho các phần tử lặp lại tham gia vào A). Lấy ví dụ, tập đa $A = \{1, 1, 2, 3\}$ của 4 phần tử. Chúng ta quan tâm đến việc tạo ra tất cả các hoán vị có lặp lại. Vì vậy, một lần nữa chúng ta đang tìm kiếm tất cả các n -bộ có thứ tự khác nhau được tạo thành bởi các phần tử của A . Lưu ý rằng khi các vị trí thay đổi các phần tử giống nhau - ví dụ: hai đầu tiên trong hoán vị $(1, 1, 2, 3)$, không mới hoán vị xảy ra. Nói chung, hai hoán vị được coi là khác nhau khi có một vị trí trong đó các phần tử khác nhau được đặt.

Tập hợp tất cả các hoán vị có lặp lại được ký hiệu là $\tilde{P}_n^{s_1, s_2, \dots, s_k}$, trong đó s_i là số phần tử loại i ($i = 1, 2, \dots, k$). Số tất cả các hoán vị có lặp lại được cho bởi công thức:

$$|\tilde{P}_n^{s_1, s_2, \dots, s_n}| = \frac{n!}{s_1! s_2! \dots s_k!}, \quad n = \sum_{i=1}^k s_i. \quad (1.13)$$

Với $s_1 = 2, s_2 = 1, s_3 = 1$ là số cần thiết $|\tilde{P}_4^{2,1,1}|$ sẽ bằng $\frac{4!}{2!1!1!} = 12$.

Bài tập

- ▷ 1.76. Viết hoán vị các phần tử của tập đa $\{1, 1, 2, 3\}$.

- ▷ 1.77. Viết chương trình tìm tất cả các hoán vị có lặp. Có thể sử dụng việc thực hiện hoán vị mà không lặp lại làm cơ sở không? Nếu vậy, những thay đổi nào là cần thiết. Nếu không - tại sao?
- ▷ 1.78. Viết chương trình tính số hoán vị có lặp với các tham số liên quan đã cho. Sử dụng thừa số hóa.
- ▷ 1.79. Để thực hiện các thuật toán mã hóa và giải mã hoán vị có lặp lại.
- ▷ 1.80. Chứng minh công thức (1.13).

1.9. Chính hợp

1.9.1. Các dạng chính hợp và cách sinh ra

Các thuật toán sau đây mà chúng ta sẽ xem xét là để tạo ra các biến thể có và không lặp lại. Mọi lập trình viên phải viết các vòng lặp lồng nhau - khi chúng là hai, ba hoặc một số xác định trước, thật dễ dàng. Tuy nhiên, khi số của chúng không được biết trước, ví dụ bài toán chúng ta đang giải yêu cầu n chu trình lồng nhau, thì cách tiếp cận tiêu chuẩn không thể áp dụng được. Hãy xem xét đoạn sau:

Ví dụ 1.1. Chính hợp có lặp

Chính hợp có lặp

```
for (a1 = 1; a1 <= k; a1++)
    for (a2 = 1; a2 <= k; a2++)
        for (a3 = 1; a3 <= k; a3++)
            ...
            for (an = 1; an <= k; an++)
                printf(" %u %u %u ... %u", a1, a2, a3, ..., an);
```

Kết quả của việc thực thi $n = 2$ và $k = 3$, tương đương với 2 chu kỳ lồng nhau cho i từ 1 đến 3, sẽ là:

1 1;1 2;1 3;2 1;2 2;2 3;3 1;3 2;3 3;

Ví dụ 1.2. Chính hợp không lặp

Chính hợp không lặp

```

for (a1 = 1; a1 <= k; a1++)
    for (a2 = 1; a2 <= k; a2++) if (a2 != a1)
        for (a3 = 1; a3 <= k; a3++) if ((a3 != a1) && (a3 != a2))
            ...
            for (an = 1; an <= k; an++)
                if ((an!=a1)&&(an!=a2)&&(an!=a3)&&...&&(an!=an-1))
                    printf("%u %u %u ... %u;", a1, a2, a3, ..., an);
    
```

Kết quả cho $n = 2$ và $k = 3$ bây giờ sẽ là:

1 2; 1 3; 2 1; 2 3; 3 1; 3 2;

Sự khác biệt giữa hai biến thể của thế hệ là ở thế hệ thứ hai không có số lặp lại.

Kết quả của đoạn đầu tiên là tạo ra tất cả các biến thể với sự lặp lại của n phần tử của lớp thứ k và kết quả của đoạn thứ hai - các biến thể không có sự lặp lại của n phần tử của lớp thứ k . Chúng ta sẽ định nghĩa các khái niệm chính xác hơn:

Cho là một tập hợp A với n phần tử.

Định nghĩa 1.22. Một biến thể có sự lặp lại của n phần tử của hạng thứ k được gọi là danh sách nhiều phần tử có k được tạo thành bởi các phần tử của A (không nhất thiết phải khác).

Tập hợp tất cả các biến thể có lặp lại được ký hiệu là \tilde{V}_n^k và số phần tử của nó là n^k . Chúng ta để lại phần chứng minh của phần sau cho người đọc như một bài tập dễ dàng.

Định nghĩa 1.23. Một biến thể không có sự lặp lại của n phần tử của lớp k được gọi là bất kỳ danh sách k phần tử có thứ tự nào được tạo thành bởi các phần tử của A .

Số lượng các biến thể khác nhau không lặp lại là $|V_n^k| = \frac{n!}{(n-k)!}$

Ta thấy ngay rằng với $k = n$ thì tập các biến thiên không lặp lại trùng với tập các hoán vị không lặp lại.

Chúng ta sẽ tạo ra tất cả các biến thể với sự lặp lại trên n và k đã cho, cho tập A , bao gồm các số tự nhiên từ 1 đến n . Điều này, tất nhiên, không thể được thực hiện với các vòng lặp lồng nhau, như trong hai phân đoạn ở trên. Chúng ta sẽ tập trung vào cách tiếp cận mà chúng ta đã sử dụng để tạo ra các hoán vị. Sự khác biệt sẽ là chúng ta sẽ đặt từng phần tử vào từng vị trí, chứ không phải chỉ một phần tử chưa được sử dụng cho đến nay (vì cho phép lặp lại). Điều này giúp loại bỏ sự cần thiết của một mảng used[] để đánh dấu các phần tử được sử dụng và chúng ta dễ dàng tiếp cận một chương trình tương tự như `permute.c`:

Chương trình 1.33. Chính hợp (136variate.c)

```
#include <stdio.h>
#define MAXN 100
/*chinh hop tu n lay k phan tu*/
const unsigned n = 4;
const unsigned k = 2;

int taken[MAXN];

void print(unsigned i)
{ unsigned k;
printf("(" );
for (k = 0; k <= i - 1; k++) printf("%u ", taken[k] + 1);
printf(")\n");
}

/*Hoi quy*/
void variate(unsigned i)
{ unsigned j;
/*Neu sai if (i>=k) thi return; o day (chi in print(i); neu muon co the sinh ra tat ca do dai 1,2, ..., k, khong chi co do dai k */
if (i >= k) { print(i); return; }
for (j = 0; j < n; j++) {
/* if (allowed(k)) { */
taken[i] = j;
variate(i + 1);
}
}
```

```
int main(void) {
    variate(0);
    return 0;
}
```

Kết quả thực hiện chương trình:

(1 1)
(1 2)
(1 3)
(1 4)
(2 1)
(2 2)
(2 3)
(2 4)
(3 1)
(3 2)
(3 3)
(3 4)
(4 1)
(4 2)
(4 3)
(4 4)

Bài tập

- 1.81. Để sửa đổi chương trình trên để tạo ra các biến thế mà không lặp lại.
- 1.82. 2. Viết các biến 3 phần tử của các phần tử của tập hợp $\{a, b, c, d, e\}$:
- với sự lặp lại;
 - không lặp lại.
- 1.83. Chứng minh rằng số biến đổi k phần tử có lặp lại trên một tập hợp n phần tử là n^k .
- 1.84. Chứng minh rằng số biến đổi k phần tử không lặp lại trên một tập hợp n phần tử là $n!/(N - k)!$

- ▷ 1.85. Để thực hiện một thuật toán để tạo lặp đi lặp lại các biến thể có / không lặp lại.
- ▷ 1.86. Để thực hiện các thuật toán mã hóa và giải mã các biến thể có và không có lặp lại.

1.10. Tổng bằng không

Bài toán. Cho các số a_1, a_2, \dots, a_n . Đặt các phép toán “+” và “-” giữa các số a_i và a_{i+1} , với $i = 1, 2, \dots, n - 1$ sao cho kết quả sau khi tính biểu thức thu được bằng 0.

Ví dụ, đối với các số tự nhiên từ 1 đến 8, một số giải pháp khả thi cho bài toán sau đây:

$$\begin{aligned} 1 + 2 + 3 + 4 - 5 - 6 - 7 + 8 &= 0 \\ 1 + 2 + 3 - 4 + 5 - 6 + 7 - 8 &= 0 \\ 1 + 2 - 3 + 4 + 5 + 6 - 7 - 8 &= 0 \\ 1 + 2 - 3 - 4 - 5 - 6 + 7 + 8 &= 0 \end{aligned}$$

Lời giải: Chúng ta sẽ tạo ra tất cả các loại biến thể bằng cách lặp lại $n - 1$ phần tử của lớp thứ hai, tức là tất cả các loại có thứ tự $(n - 1)$ -bộ, bao gồm 0 và 1 (tương ứng với một dấu dương và một dấu âm ở phía trước của số tương ứng). Với mỗi $(n - 1)$ -bộ như vậy, chúng ta sẽ kiểm tra xem nó có phải là một nghiệm của bài toán hay không, và với mục đích này, chúng ta sẽ tính toán giá trị của biểu thức tương ứng. Cách thực hiện đầy đủ của giải pháp này như sau, vì dữ liệu đầu vào được đặt ở đầu dưới dạng hằng số n là số lượng các số và mảng $a[]$ chứa chính các số:

Chương trình 1.34. Tổng bằng 0 (137sumzero.c)

```
#include <stdio.h>
#include <math.h>

/* So luong so trong day */
const unsigned n = 8;
/* Day so */
int a[] = { 1, 2, 3, 4, 5, 6, 7, 8 };
/* Tim tong */
```

```
int sum = 0;

void checkSol(void)
{ unsigned i;
    int tempSum = 0;
    for (i = 0; i < n; i++) tempSum += a[i];
    if (tempSum == sum) /* Tim duoc nghiem => In no ra*/
        for (i = 0; i < n; i++)
            if (a[i] > 0) printf("+%d ", a[i]);
            else printf("%d ", a[i]);
        printf(" = %d\n", tempSum);
    }
}

void variate(unsigned i)
{ if (i >= n) {
    checkSol();
    return;
}
    a[i] = abs(a[i]); variate(i + 1);
    a[i] = -abs(a[i]); variate(i + 1);
}

int main(void) {
    variate(0);
    return 0;
}
```

Kết quả thực hiện chương trình:

$$\begin{aligned}
 +1 + 2 + 3 + 4 - 5 - 6 - 7 + 8 &= 0 \\
 +1 + 2 + 3 - 4 + 5 - 6 + 7 - 8 &= 0 \\
 +1 + 2 - 3 + 4 + 5 + 6 - 7 - 8 &= 0 \\
 +1 + 2 - 3 - 4 - 5 - 6 + 7 + 8 &= 0 \\
 +1 - 2 + 3 - 4 - 5 + 6 - 7 + 8 &= 0 \\
 +1 - 2 - 3 + 4 + 5 - 6 - 7 + 8 &= 0 \\
 +1 - 2 - 3 + 4 - 5 + 6 + 7 - 8 &= 0 \\
 -1 + 2 + 3 - 4 + 5 - 6 - 7 + 8 &= 0 \\
 -1 + 2 + 3 - 4 - 5 + 6 + 7 - 8 &= 0 \\
 -1 + 2 - 3 + 4 + 5 - 6 + 7 - 8 &= 0 \\
 -1 - 2 + 3 + 4 + 5 + 6 - 7 - 8 &= 0 \\
 -1 - 2 + 3 - 4 - 5 - 6 + 7 + 8 &= 0 \\
 -1 - 2 - 3 + 4 - 5 + 6 - 7 + 8 &= 0 \\
 -1 - 2 - 3 - 4 + 5 + 6 + 7 - 8 &= 0
 \end{aligned}$$

Bài tập

- 1.87. Để giải bài toán, nếu trong phép cộng và phép trừ, có thể đặt một dấu nhân.
- 1.88. Đề xuất một cách để lưu một số phép tính. Ví dụ, khi quá trình tạo có thể bị gián đoạn sau vị trí thứ k , i . tiêu chí nào có thể đảm bảo rằng tổng 0 sẽ không nhận được, bất kể phép tính số học nào sẽ được đặt trên $n - k$ vị trí khác?

1.11. Tổ hợp

Nếu cho đến nay thứ tự của các yếu tố đã quan trọng, thì trong đoạn này nó sẽ không còn khiến chúng ta quan tâm nữa. Cho một tập hợp A có n phần tử.

Định nghĩa 1.24. Một tổ hợp không có sự lặp lại của n phần tử thuộc lớp k được gọi là tập hợp con k phần tử của A .

Định nghĩa 1.25. Một tổ hợp có sự lặp lại của n phần tử của lớp k được gọi là một tập hợp nhiều phần tử có chứa các phần tử của A .

Ví dụ: đối với $n = 4, k = 2$ của tập bốn phần tử a, b, c, d thì các kết hợp không có sự lặp lại của lớp thứ hai là $\{a, b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}, \{c, d\}$ và các kết hợp có lặp lại là:

$\{a, a\}, \{a, b\}, \{a, c\}, \{a, d\}, \{b, b\}, \{b, c\}, \{b, d\}, \{c, c\}, \{c, d\}, \{d, d\}$.

Tổ hợp được sử dụng rộng rãi trong các bài toán tối ưu hóa. Trong tương lai chúng ta sẽ gặp chúng nhiều lần, vì vậy các thuật toán cho thế hệ của chúng rất quan trọng. Một số bài toán liên quan đến tổ hợp được đưa ra trong phần 1.5.

Số lượng các kết hợp không có sự lặp lại, mà chúng ta đã thấy cách chúng ta có thể tính toán bằng cách tính toán trong 1.1.5., Là

$$C_n^k = C_n^k = \frac{n(n-1)...(n-k+1)}{k(k-1)...1} = \frac{n!}{k!(n-k)!}$$

và các tổ hợp có lặp lại:

$$\tilde{C}_n^k = \frac{(n+k-1)!}{k!(n-1)!}.$$

Việc tạo các tổ hợp không lặp lại sẽ được thực hiện theo sơ đồ của thuật toán tạo hoán vị. Chúng ta tuân thủ nó, bởi vì khi được tạo theo thứ tự từ điển, sẽ cực kỳ thuận tiện nếu bạn cần "cắt" một số giải pháp đã tạo. Như đã đề cập, sự cắt giảm như vậy xảy ra khi giải quyết các vấn đề thực tế, trong đó việc nghiên cứu một nhánh lớn của các cấu hình tổ hợp có thể là vô nghĩa.

Chương trình 1.35. Tìm tổ hợp (138comb.c)

```
#include <stdio.h>
#define MAXN 20
/*Tìm tất cả các tổ hợp của n phần tử của lớp k*/
const unsigned n = 5;
const unsigned k = 3;
unsigned mp[MAXN];
void print(unsigned length)
{
    unsigned i;
    for (i = 0; i < length; i++) printf("%u ", mp[i]);
    printf("\n");
}
void comb(unsigned i, unsigned after)
```

```

{ unsigned j;
if (i > k) return;
for (j = after + 1; j <= n; j++) {
    mp[i - 1] = j;
    if (i == k) print(i);
    comb(i + 1, j);
}
int main() {
    printf("C(%u,%u): \n", n, k);
    comb(1, 0);
    return 0;
}

```

Kết quả thực hiện chương trình:

```

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 tập

- ▷ 1.89. Để sửa đổi chương trình trên để nó tạo ra các kết hợp có lặp lại.
- ▷ 1.90. Viết các tổ hợp 3 phần tử của các phần tử của tập hợp $\{a, b, c, d, e\}$:
 - với sự lặp lại
 - không lặp lại
- ▷ 1.91. Giải thích mối quan hệ giữa số tổ hợp không lặp lại của n phần tử hạng k và hệ số của nhị thức.

- ▷ 1.92. Chứng minh rằng số tổ hợp k phần tử không lặp lại trên một tập n phần tử là $n!/(k!(n-k)!)$.
- ▷ 1.93. Chứng minh rằng số tổ hợp gồm k phần tử có lặp lại trên một tập n phần tử là $(n+k-1)!/(k!(n-1)!)$. Đưa ra hai chứng minh: một đại số và một tổ hợp.
- ▷ 1.94. Để thực hiện một thuật toán để tạo lặp đi lặp lại các kết hợp có / không lặp lại.
- ▷ 1.95. Để thực hiện các thuật toán mã hóa và giải mã kết hợp có / không lặp lại.

1.12. Biểu diễn số thành tổng

1.12.1. Tạo ngắt số dưới dạng tổng của các số đã cho

Bài toán: Tìm tất cả các biểu diễn không có thứ tự (phân tích) có thể có của n trên một số tự nhiên n đã cho dưới dạng tổng các số tự nhiên (không nhất thiết khác nhau). Ví dụ, số 5 có thể được chia theo 7 cách sau:

$$\begin{aligned} 5 &= 5 \\ 5 &= 4 + 1 \\ 5 &= 3 + 2 \\ 5 &= 3 + 1 + 1 \\ 5 &= 2 + 2 + 1 \\ 5 &= 2 + 1 + 1 + 1 \\ 5 &= 1 + 1 + 1 + 1 + 1 \end{aligned}$$

Thuật toán mà chúng ta sẽ nhận ra sự phân rã là đệ quy:

$$\text{devNum}(0) = \{\}$$

$$\text{devNum}(n) = \{k\} + \text{devNum}(n - k), k = n, n - 1, \dots, 1.$$

Chúng ta phải cẩn thận và tránh tạo ra các lỗi lặp đi lặp lại, chẳng hạn như

$$5 = 3 + 2$$

$$5 = 2 + 3$$

Cách sau rất dễ thực hiện: chúng ta sẽ yêu cầu mỗi phép cộng tiếp theo nhỏ hơn hoặc bằng lần trước. Hàm đệ quy thực hiện devNum có hai đối số: n (số break) và pos - một biến cho biết số lần bị phá vỡ cho đến nay:

Chương trình 1.36. Biểu diễn thành tổng số (139devnum.c)

```
#include <stdio.h>
#define MAXN 100
const unsigned n = 7;
unsigned mp[MAXN + 1];
void print(unsigned length)
{
    unsigned i;
    for (i = 1; i < length; i++)
        printf("%u+", mp[i]);
    printf("%u\n", mp[length]);
}
void devNum(unsigned n, unsigned pos)
{
    if (0 == n)
        print(pos-1);
    else {
        unsigned k;
        for (k = n; k >= 1; k--) {
            mp[pos] = k;
            if (mp[pos] <= mp[pos-1])
                devNum(n-k, pos+1);
        }
    }
}
int main()
{
    mp[0] = n+1;
    devNum(n, 1);
    return 0;
}
```

Kết quả thực hiện chương trình:

7

6+1

5+2

5+1+1

4+3
 4+2+1
 4+1+1+1
 3+3+1
 3+2+2
 3+2+1+1
 3+1+1+1+1
 2+2+2+1
 2+2+1+1+1
 2+1+1+1+1+1
 1+1+1+1+1+1+1

Nếu nhiệm vụ chỉ tìm kiếm số lần ngắt khác nhau, thì (như với các cấu hình tổ hợp khác) không cần thiết phải tạo và đếm chúng hoàn toàn. Một cách tiếp cận như vậy sẽ cực kỳ không hiệu quả. Có một công thức lặp lại cho phép tính trực tiếp con số này. Trên thực tế, những thứ với công thức này không đơn giản như vậy và cần phải sử dụng tính năng tối ưu hóa động, vì vậy chúng ta sẽ xem xét nó ở phần sau, trong 8.3.6.

Bài tập

- ▷ 1.96. Để triển khai một thuật toán tạo ngắt theo thứ tự từ vựng.
- ▷ 1.97. Để triển khai một phiên bản lặp đi lặp lại của thuật toán để tạo ra sự cố.

1.12.2. Sinh ra tất cả biểu diễn một số như là tích của các số tự nhiên

Sự khác biệt với đoạn trước, cả về thuật toán và cách triển khai, là tối thiểu. Thay vì devNum ($n-k$, $cnt + 1$), chúng ta sẽ gọi đệ quy devNum (n / k , $cnt + 1$), không phải cho mọi k , mà chỉ cho những người có $n \% k == 0$. Điều kiện để tiếp tục phân tích (vòng lặp for) sẽ là $k > 1$, không phải $k \geq 1$, tức là phần dưới cùng của đệ quy sẽ là $k == 1$, không phải $k == 0$ (sau này dễ dàng giải thích: 0 và 1 là đồng dạng của các phép toán cộng và nhân). Sau đây là cách triển khai đã sửa đổi:

Chương trình 1.37. Biểu diễn thành tích số (140devnum2.c)

```
#include <stdio.h>
#define MAXLN 20 /* Hệ số: tối đa log2n (tối thiểu 2) */
const unsigned n = 50; /* Một con số chúng ta sẽ tách ra */
unsigned mp[MAXLN];
void print(unsigned length)
{ unsigned i;
    for (i = 1; i < length; i++) printf("%u * ", mp[i]);
    printf("%d\n", mp[length]);
}
void devNum(unsigned n, unsigned pos) {
if (1 == n)
    print(pos-1);
else {
    unsigned k;
    for (k = n; k > 1; k--) {
        mp[pos] = k;
        if (mp[pos] <= mp[pos-1] && n % k == 0)
            devNum(n / k, pos+1);
    }
}
int main() {
    mp[0] = n + 1;
    devNum(n, 1);
    return 0;
}
```

Kết quả thực hiện chương trình:

50
25 * 2
10 * 5
5 * 5 * 2

Bài tập

- ▷ 1.98. Để thực hiện một thuật toán để tạo ra các ngắt theo thứ tự từ vựng.
- ▷ 1.99. Để triển khai một phiên bản lặp đi lặp lại của thuật toán để

tạo ra sự cỗ.

1.12.3. Sinh ra tất cả biểu diễn một số như là tổng của các số tự nhiên

Thú vị hơn là trường hợp chúng ta phải chia một số tự nhiên là tổng của các số tự nhiên đã cho. Ví dụ, chúng ta có tem bưu chính BGN 2, 5 và 10 và chúng ta phải gửi một bưu kiện có giá trị BGN 20. Tất cả các khả năng (tổng cộng là 6) để hình thành số tiền này là:

$$20 = 10 + 10$$

$$20 = 10 + 5 + 5$$

$$20 = 10 + 2 + 2 + 2 + 2 + 2$$

$$20 = 5 + 5 + 5 + 5$$

$$20 = 5 + 5 + 2 + 2 + 2 + 2 + 2$$

$$20 = 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2$$

Thuật toán giải bài toán này tương tự như thuật toán ngắn một số dưới dạng tổng các số tự nhiên. Hãy để các số chúng ta có thể sử dụng trong bảng phân tích nằm trong mảng given[gN]. Chúng ta sẽ thực hiện một chu trình cho $p = 0, 1, \dots, gN-1$, và trong lệnh gọi đệ quy thay vì p , chúng ta sẽ giảm n bởi giá trị tương ứng của given[p]:

Chương trình 1.38. Biểu diễn thành tích số (141devnum3.c)

```
#include <stdio.h>
#define MAX_ADDS 100
/* Tổng sẽ biểu diễn*/
const unsigned n = 15;
/* Số giá trị đồng xu khác nhau */
const unsigned gN = 3;
/* Giá trị của đồng xu */
const unsigned given[] = { 2, 3, 5 };
unsigned mp[MAX_ADDS];
void print(unsigned length)
{
    unsigned i;
    for (i = 1; i < length; i++)
        printf("%u + ", mp[i]);
    printf("%d\n", mp[length]);
```

```

    }
void devNum(unsigned n, unsigned pos)
{ unsigned k, p;
  for (p = gN; p > 0; p--) {
    k = given[p - 1];
    if (n > k) {
      mp[pos] = k;
      if (mp[pos] <= mp[pos - 1])
        devNum(n - k, pos + 1);
    }
    else if (n == k) {
      mp[pos] = k;
      if (mp[pos] <= mp[pos - 1])
        print(pos);
    }
  }
  int main() {
    mp[0] = n + 1;
    devNum(n, 1);
    return 0;
}

```

Kết quả thực hiện chương trình:

5 + 5 + 5
 5 + 5 + 3 + 2
 5 + 3 + 3 + 2 + 2
 5 + 2 + 2 + 2 + 2 + 2
 3 + 3 + 3 + 3 + 3
 3 + 3 + 3 + 2 + 2 + 2
 3 + 2 + 2 + 2 + 2 + 2 + 2

Bài tập

- ▷ **1.100.** Có thể tổng quát thuật toán thành số thực không? Nếu không: tại sao? Nếu có: những thay đổi nào là cần thiết?
- ▷ **1.101.** Để thực hiện một thuật toán để tạo ra các ngắt từ vựng.
- ▷ **1.102.** Để triển khai một biến thể lặp đi lặp lại của thuật toán tạo ra sự cố.

► **1.103.** Đề xuất và thực hiện thuật toán ngắn một số dưới dạng tích của các số đã cho.

1.12.4. Phân hoạch một tập hợp

Nhiệm vụ chúng ta sẽ xem xét là tìm số lần ngắn có thể có của một tập hợp, tức là biểu diễn của tập hợp như một hợp của các tập hợp con không rỗng không giao nhau của nó. Ví dụ, đối với tập hợp $A = \{1, 2, 3\}$ các ngắn sẽ là:

- $\{1, 2, 3\}$
- $\{1, 2\}, \{3\}$
- $\{1, 3\}, \{2\}$
- $\{1\}, \{2, 3\}$
- $\{1\}, \{2\}, \{3\}$

Số Bell và Stirling

Mặc dù có một số công thức trực tiếp cho số lượng cấu hình tổ hợp được thảo luận ở trên, nhưng ở đây mọi thứ phức tạp hơn một chút.

Số lần ngắn cho một tập hợp có n phần tử bằng số thứ n của Bell $B(n)$. Số Bell được định nghĩa như sau:

$$B(n) = \sum_{i=0}^n St(n, k),$$

trong đó $St(n, k)$ là số chuỗi của loại thứ hai, được định nghĩa đê quy như sau:

$$St(n, k) = \begin{cases} St(n - 1, k - 1) + kSt(n - 1, k) & \text{với } k = 1, 2, \dots, n; \\ 1 & \text{với } n > 0, k - 1; \\ 0 & \text{với } k = 0; \\ 0 & \text{với } n = 0. \end{cases}$$

Số Stirling đại diện cho số lượng biểu diễn của một tập hợp n phần tử như một hợp của chính xác k tập hợp con không rỗng của nó.

Chúng ta có thể tính toán số Stirling theo một số cách. Một là sử dụng trực tiếp định nghĩa đê quy và thực hiện một hàm đê quy tương ứng. Tuy nhiên, một số giá trị sẽ được tính toán nhiều lần,

đây là một giải pháp không hiệu quả. Chúng ta đã gặp phải vấn đề tương tự khi tính toán chuỗi Fibonacci. Chúng ta sẽ tìm các số Stirling và Bell theo cách lặp: Đầu tiên chúng ta sẽ tìm các số Stirling $M[i]=St(n, i)$, $i = 0, 1, \dots, n$. Sau đó, chúng ta sẽ tìm số Bell thứ n là tổng của các số Stirling tương ứng.

Số Stirling xác định một tam giác tương tự như tam giác Stirling của Pascal. Quan sát này trực tiếp dẫn chúng ta đến ý tưởng nhận ra một thuật toán tương tự như một thuật toán mà chúng ta đã sử dụng cho các hệ số nhị thức.

			1				
			1	1	1		
			1	3	1		
			1	7	6	1	
			1	15	25	10	1
			1	31	90	65	15
		
		

Hình 1.15. Tam giác Stirling

Chương trình 1.39. Số phân hoạch tập hợp (142bell.c)

```
#include <stdio.h>
#define MAXN 100
const unsigned long n = 10;
unsigned long M[MAXN+1];
void stirling (unsigned n)
{
    unsigned i, j;
    if (0 == n) M[0] = 1;
    else M[0] = 0;
    for (i = 1; i <= n; i++) {
        M[i] = 1;
        for (j = i-1; j >= 1; j--) M[j] = j*M[j]+M[j-1];
    }
}
unsigned long bell(unsigned n)
{
    unsigned i;
    unsigned long result = 0;
    for (i = 0; i <= n; i++) result += M[i];
    return result;
}
```

```

}
int main() {
    stirling (n);
    printf("bell(%lu) = %lu\n", n, bell (n));
    return 0;
}

```

Kết quả thực hiện chương trình:

`bell(10) = 115975`

Bài tập

- ▷ **1.104.** Tìm số Stirling thứ năm theo cách thủ công.
- ▷ **1.105.** Để triển khai một biến thể đệ quy của thuật toán tìm số Stirling.
- ▷ **1.106.** Tìm ra các công thức phản ánh mối quan hệ giữa các số trong tam giác Stirling. So sánh với những gì trong tam giác Pascal của 1.1.5.
- ▷ **1.107.** Viết một chương trình để tìm tất cả các điểm ngắt của một tập hợp đã cho dưới dạng hợp nhất của các tập hợp con không rỗng của nó.

1.13. Đánh giá và độ phức tạp của thuật toán

Chúng ta sẽ kết thúc chương mở đầu này với phần giới thiệu ngắn gọn về bộ máy toán học cần thiết cho việc nghiên cứu đầy đủ bất kỳ thuật toán máy tính nào. Chủ đề đánh giá và độ phức tạp của các thuật toán là quan trọng và không nên bỏ qua. Các chỉ định và đặc tính mà chúng ta sẽ trình bày sẽ được tìm thấy gần như liên tục trong tài liệu bên dưới.

Khi xem xét một thuật toán máy tính, chúng ta thường quan tâm đến ba thuộc tính của nó:

- đơn giản (và sang trọng)
- tính đúng đắn
- tốc độ

Trong khi cái đầu tiên trong số này có thể được "đo lường" bằng trực giác (và hơi chủ quan), hai cái sau đòi hỏi phân tích chuyên

sâu hơn nhiều. Các kỹ năng chúng ta sẽ có được trong đoạn này sẽ cho phép chúng ta xác định dễ dàng và chính xác hiệu quả của một thuật toán, so sánh các thuật toán, để tăng tốc độ của chúng thông qua các thay đổi được đo lường và chính xác.

Hãy xem đoạn chương trình sau:

Các dòng lệnh tính toán

- 1) `n = 100;`
- 2) `sum = 0;`
- 3) `for (i = 0; i < n; i++)`
- 4) `for (j = 0; j < n; j++)`
- 5) `sum++;`

Chúng ta quan tâm đến việc chương trình trên sẽ hoạt động nhanh như thế nào. Những gì chúng ta có thể làm bằng thực nghiệm là kiểm tra xem mất bao lâu để hoàn thành công việc của nó. Để nghiên cứu hành vi của nó một cách tổng quát hơn, chúng ta có thể thực hiện nó cho các giá trị khác của n . Bảng 1.5. cho thấy mối quan hệ giữa lượng dữ liệu đầu vào và tốc độ thực thi.

Kích thước đầu vào	Thời gian thực hiện
10	0,000001 giây
100	0,0001 giây
1000	0,01 giây
10000	1,071 giây
100000	106,543 giây
1000000	10663,6 giây

Bảng 1.5. sự phụ thuộc giữa kích thước dữ liệu đầu vào và tốc độ thực thi.

Từ Bảng 1.5 ta thấy rằng khi ta tăng n lên mươi lần thì thời gian thực hiện của chương trình tăng lên 100 lần.

Chúng ta hãy xem xét phân đoạn trên sâu hơn. Dòng 1) và 2) có một khởi tạo tĩnh mất một thời gian không đổi. Hãy để chúng ta biểu thị nó bằng a . Đối với các hoạt động $i = 0$ và $i ++$, cũng như đối với việc kiểm tra $i < n$, một lần nữa yêu cầu thời gian không đổi (mỗi trong số chúng đại diện cho một số lượng lệnh không đổi của bộ xử

lý), chúng ta sẽ ký hiệu nó bằng b , c , d , tương ứng. Trong dòng 4) thời gian cần thiết cho các phép toán $j = 0$, $j < n$ và $j ++$ được ký hiệu là e , f , g . Cuối cùng, hoạt động trên dòng 5) cũng yêu cầu một thời gian không đổi: giả sử là h .

Với các ký hiệu được giới thiệu theo cách này, không khó để tính tổng thời gian hoạt động của chương trình cho một giá trị tùy ý của n :

$$\begin{aligned} a + b + n.c + n.d + n.(e + n.f + n.g + n.h) &= \\ = a + b + n.c + n.d + n.e + n.n.f + n.n.g + n.n.h &= \\ = n^2.(f + g + h) + n.(c + d + e)a + b \end{aligned}$$

Nhớ lại rằng a, b, c, d, e, f, g, h là các hằng số.

Hãy biểu thị:

$$\begin{aligned} i &= f + g + x \\ j &= c + d + e \\ k &= a + b \end{aligned}$$

(ở đây i và j không liên quan gì đến các biến được sử dụng trong đoạn trên). Do đó, thuật toán được thực thi trong thời gian:

$$i.n^2 + j.n + k$$

Các hằng số i, j và k quan trọng đối với tốc độ của thuật toán, nhưng không có ý nghĩa quyết định. Trong thực tế, khi chúng ta nghiên cứu tính hiệu quả của một thuật toán, chúng ta không quan tâm đến chúng. Các hằng số này phụ thuộc chủ yếu vào hiệu suất máy của chương trình của chúng ta, cũng như tốc độ của máy mà nó chạy.

Hơn nữa, khi chúng ta nghiên cứu hành vi của thuật toán của mình, chúng ta có thể bỏ qua ngay cả các đơn thức $j.n$ và k và chỉ để lại một đơn thức trong đó n tham gia nhiều nhất. Mục đích của việc "đơn giản hóa" này là chỉ để lại tính năng quan trọng nhất cho một thuật toán nhất định, tức là chức năng mà thời gian thực thi phụ thuộc ở mức độ lớn nhất, tức là phát triển nhanh nhất khi lượng dữ liệu đầu vào tăng lên.

Hãy xem xét hai hàm cho biết thời gian thực hiện của hai thuật toán A_1 và A_2 đã cho phụ thuộc vào kích thước n của dữ liệu đầu vào: $f = 2n^2$ và $g = 200n$. Để dàng nhận thấy rằng mặc dù hệ số của g lớn hơn rất nhiều so với của f nhưng khi n vượt qua một giá trị cố định nào đó (trong trường hợp này là $n > 100$) thì thuật toán A_2 sẽ giải bài toán nhanh hơn A_1 . Hơn nữa, n càng tăng thì mối quan hệ giữa thời gian thực hiện của hai thuật toán càng tăng theo hướng có lợi cho A_2 . Về mặt tiệm cận, thuật toán A_2 nhanh hơn và độ phức tạp của nó là tuyến tính, trong khi thuật toán A_1 là bậc hai.

Trong khai báo dưới đây, chúng ta sẽ dựa trên những quan sát gần đây của mình trên những nền tảng vững chắc hơn.

Bài tập

- ▷ **1.108.** Cho ba thuật toán có độ phức $5n^2 - 7n + 13$, $3n^2 + 15n + 100$ và $1000n$. Cái nào trong số chúng nên được sử dụng cho dữ liệu đầu vào lên đến: 100; 1000; 10000; 1000000?
- ▷ **1.109.** Có thể cho đa thức $5n^3 - 5n^2 + 5$ xác định độ phức tạp của một thuật toán không? Và đa thức $5n^3 - 5n^2 - 5$? Còn $-5n^3 - 5n^2 + 5$ thì sao?

1.13.1. Lượng dữ liệu đầu vào

Giả sử một bài toán trong đó kích thước của dữ liệu đầu vào được xác định bởi một số nguyên n . Hầu hết tất cả các nhiệm vụ chúng ta sẽ xem xét có thuộc tính này. Chúng ta sẽ làm rõ vấn đề sau bằng cách xem xét một số ví dụ:

Ví dụ 1.3. Một mảng có n phần tử được cho và chúng ta muốn sắp xếp nó. Kích thước của dữ liệu đầu vào được xác định bởi số n phần tử của mảng.

Ví dụ 1.4. Hai số tự nhiên a và b đã cho, ta tìm ước chung lớn nhất của chúng. Ở đây kích thước của dữ liệu đầu vào được xác định bởi số lượng các chữ số nhị phân (bit) lớn hơn trong số các số a và b : tức là từ $\log_2(\max(a, b))$. Khi chúng ta giới thiệu bộ máy nghiên cứu tính hiệu quả của các thuật toán dưới đây, chúng ta sẽ thấy lý do tại sao chúng ta chọn cực đại của hai số.

Ví dụ 1.5. Một đồ thị được đưa ra và chúng ta tìm cây bao trùm của nó (xem 5.1.).

Trong trường hợp này, rất thuận tiện để mô tả kích thước của đầu vào với hai tham số: số đỉnh và số cạnh của đồ thị.

1.13.2. Ký hiệu tiệm cận

Khi chúng ta quan tâm đến độ phức tạp của một thuật toán, chúng ta thường quan tâm đến việc nó hoạt động như thế nào ở kích thước đủ lớn n của dữ liệu đầu vào. Khi chính thức ước tính độ phức tạp của các thuật toán, chúng ta sẽ quan tâm đến hành vi của chúng tại n , có xu hướng đến vô cùng. Chúng ta sẽ mô tả độ phức tạp của một thuật toán với các hàm có dạng $f : \mathbb{N} \rightarrow \mathbb{N}$. (Nhớ lại rằng \mathbb{N} biểu thị tập các số tự nhiên: $0, 1, 2, \dots$). Đôi khi chúng ta sẽ làm việc với các hàm được xác định trên một tập con của \mathbb{N} , chẳng hạn chỉ hợp lệ với n chẵn hoặc từ một giá trị nhất định trở đi. Chúng ta cũng sẽ sử dụng các hàm thực, ví dụ $\log n$, và về nguyên tắc, chúng ta sẽ ngụ ý hạn chế của chúng đối với \mathbb{N} . Các trường hợp sau không phải là rất "thuần túy" theo quan điểm lý thuyết, nhưng chúng tiết kiệm được rất nhiều khó khăn.

Định nghĩa 1.26. $O(F(n)) = \{f(n) | \exists c(c > 0), \exists n_0(c) : \forall n > n_0 : 0 \leq f(n) \leq c.F(n)\}$

Nghĩa là $O(F(n))$ là tập các hàm f trong đó có một hằng số $c(c > 0)$ sao cho $f(n) \leq c.F(n)$, với mọi giá trị đủ lớn của n , tức là tồn tại một hằng số n_0 (có thể phụ thuộc vào c) mà bất đẳng thức trên áp dụng với mọi $n > n_0$. Do đó $O(F)$ xác định tập hợp tất cả các hàm phát triển không nhanh hơn F .

Khi xem xét độ phức tạp của các thuật toán, hai ký hiệu cơ bản hơn được sử dụng: $\Theta(F)$ và $\Omega(F)$.

Định nghĩa 1.27. $\Omega(F(n)) = \{f(n) | \exists c(c > 0), \exists n_0(c) : \forall n > n_0 : f(n) \geq c.F(n) \geq 0\}$

Nghĩa là, $\Omega(F)$ là tập các hàm $f(n)$ mà $f(n) \geq c.F(n)$ với mọi $n > n_0$. Do đó $\Omega(F)$ bao gồm tất cả các hàm tăng không chậm hơn F .

Định nghĩa 1.28. $\Theta(F(n)) = \{f(n) | \exists c_1(c_1 > 0), \exists c_2(c_2 > 0), \forall n_0(c_1, c_2) : \forall n \geq n_0 : 0 \leq c_1.F(n) \leq f(n) \leq c_2.F(n)\}$

Nó dựa trực tiếp từ định nghĩa rằng $\Theta(F) = O(F) \cap \Omega(F)$. Nghĩa là, $\Theta(F)$ chứa tất cả các hàm phát triển nhanh như F (lên đến một cấp số nhân không đổi).

Các ký hiệu sau ít được sử dụng hơn (lưu ý sự bất đẳng thức nghiêm ngặt trong định nghĩa.):

Định nghĩa 1.29. $o(F(n)) = \{f(n) | \forall c(c > 0), \exists n_0(c) : n > n_0 : 0 \leq f(n) < c.F(n)\}$

Nó dựa trực tiếp từ định nghĩa rằng $o(F) = O(F) \setminus \Theta(F)$. Nghĩa là $o(F)$ chứa tất cả các hàm phát triển chậm hơn F (lên đến một cấp số nhân không đổi).

Lưu ý rằng trong khi $3n^2 \in O(n^2)$ và $3n^2 \in O(n^3)$, thì $3n^2 \in o(n^3)$, nhưng $3n^2 \neq o(n^2)$.

Định nghĩa 1.30. $\omega(F(n)) = \{f(n) | \forall c(c > 0), \exists n_0(c) : n > n_0 : 0 \leq c.F(n) < f(n)\}$

Nó dựa trực tiếp từ định nghĩa rằng $\omega(F) = \Omega(F) \setminus \Theta(F)$. Nghĩa là, $\omega(F)$ chứa tất cả các hàm phát triển nhanh hơn F (lên đến một cấp số nhân không đổi).

Trong các phần tiếp theo, chúng ta sẽ xem xét các thuộc tính hữu ích và ví dụ về cách chúng ta có thể đánh giá độ phức tạp của một thuật toán trong thực tế.

Dưới đây chúng ta sẽ sử dụng ký hiệu chuẩn để chỉ ra thuộc tập " \in " để chỉ ra rằng $f \in \xi(F)$, trong đó ξ là một trong những hàm tiệm cận trên. Tuy nhiên, khi nào thuận tiện, chúng ta sẽ thay bằng dấu bằng. Mặc dù điều này thoát nghe có vẻ lạ, nhưng nó mang lại cho chúng ta lợi thế là có thể viết các phụ thuộc lặp lại cho hàm tiệm cận $T(n)$ có dạng:

$$T(n) = T(n - 1) + O(n)$$

Bài tập

- ▷ 1.110. Chứng minh rằng $\Theta(F) = O(F) \cap \Omega(F)$.

- 1.111. Chứng minh rằng $o(F) = O(F) \setminus \Theta(F)$.
- 1.112. Chứng minh rằng $\omega(F) = \Omega(F) \setminus \Theta(F)$.

1.13.3. Tính chất và ví dụ của $O(F)$

Cho đến gần đây, ký hiệu $O(F)$ được sử dụng phổ biến nhất để đánh giá độ phức tạp của các thuật toán và chương trình. Gần đây, một ước tính chính xác hơn về $\Theta(F)$ đã được ưa chuộng hơn, nhưng điều này đòi hỏi nỗ lực bổ sung trong việc phân tích các thuật toán và vẫn chưa phải là một cách tiếp cận được chấp nhận rộng rãi. Trong các chương tiếp theo, chúng ta thường sử dụng ký hiệu $\Theta(\dots)$ và chỉ trong các trường hợp đặc biệt - các ký hiệu khác.

Một số tính chất của $O(F)$:

1. Tính phản xạ: $f \in O(f)$
2. Tính bắc cầu: $f \in O(g), g \in O(h) \Rightarrow f \in O(h)$
3. Tính đối xứng chuyển vị: $f \in \Omega(g) \Leftrightarrow g \in O(f)$
4. Có thể bỏ qua các hằng số: Với mỗi $k > 0, k.F \in O(F)$.
5. n , nâng cao hơn, phát triển nhanh hơn: $n^r \in O(n^s)$, với $0 \leq r \leq s$.
6. Sự gia tăng của tổng các hàm được xác định bởi tiệm cận nhanh nhất tăng dần (điều này được biểu thị bằng max):

$$f + g \in O(\max(f, g)),$$

có thể được viết như sau:

$$f \in O(g) \Rightarrow f + g \in O(g)$$

hoặc là:

$$O(c_1 f + c_2 g) \in O(\max(f, g)), \text{ với } c_1, c_2 > 0.$$

7. Nếu $f(n)$ là đa thức bậc d trở xuống thì $f \in O(n^d)$.
8. Tích của các hàm số:

$$f \in O(F) \text{ và } g \in O(G) \Rightarrow f.g \in O(F.G)$$

Ta có thể chứng minh theo định nghĩa các tính chất trên:

Để khẳng định rằng hàm số f thuộc $O(g)$, cần và đủ để tìm một số tự nhiên n_0 và một hằng số dương c sao cho với mọi tự nhiên $n(n > n_0)$ thì bất phương trình $f(n) \leq cg(n)$.

Tính chất tiệm cận (giả sử tồn tại giới hạn):

1. $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \in \mathbb{R}^* \Rightarrow f(n) \in O(g(n)), g(n) \in O(f(n))$.
2. $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in O(g(n)), g(n) \notin O(f(n))$.
3. $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) \notin O(g(n)), g(n) \in O(f(n))$.

Các thuộc tính này có thể được sử dụng như một cách thay thế để xác minh rằng một hàm thuộc nhóm các hàm kiểu $O(\dots)$.

Ví dụ 1.6. Xét các hàm $\log n$ và \sqrt{n} . Sử dụng quy tắc Lopital, nhận được

$$\lim_{n \rightarrow \infty} \frac{\log n}{\sqrt{n}} = \lim_{n \rightarrow \infty} \frac{1/n}{1/(2\sqrt{n})} = \lim_{n \rightarrow \infty} \frac{2}{\sqrt{n}} = 0.$$

Phương trình cuối cùng suy ra là $\log n \in O(\sqrt{n})$, nhưng $\sqrt{n} \notin O(\log n)$.

Ví dụ 1.7. Các ví dụ khẳng định

- $10n \in O(n); \quad 10n \in O(n^2); \quad 10n \in O(n^4);$
- $10n \in O(3n^4 - 10n^2 + 7); \quad 10n + 3 \in O(n);$
- $4n^2 - 5n + 2 \in O(n^2); \quad 4n^3 + 5n^2 + 5 \in O(n^3);$
- $\sqrt{n} \in O(n); \quad \log n \in O(\sqrt{n}).$

Ví dụ 1.8. Các ví dụ phủ định

$$4n^2 - 5n + 2 \notin O(n^3); \quad \sqrt{n} \notin \Omega(n); \quad \log n \notin \Omega(\sqrt{n}).$$

Bài tập

- ▷ **1.113.** Kiểm tra theo định nghĩa rằng $5n^3 - 5n^2 + 5 \in \Omega(n^3)$.
- ▷ **1.114.** Chứng minh rằng $\sqrt{n} \in \Omega(n^{0,48})$.
- ▷ **1.115.** Chứng minh các tính chất tổng quát trên của $\Omega(F)$.
- ▷ **1.116.** Chứng minh tính chất tiệm cận trên của $\Omega(F)$.

1.13.4. Tính chất và ví dụ về Θ

Một số thuộc tính của $\Theta(F)$:

1. Tính phản xạ: $f \in \Theta(f)$
2. Tính bắc cầu: $f \in \Theta(g), g \in \Theta(h) \Rightarrow f \in \Theta(h)$
3. Tính đối xứng: $f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$
4. Có thể bỏ qua các hằng số: Với mỗi $k > 0, k.F \in \Theta(F)$.
5. Sự gia tăng của tổng các hàm được xác định bởi sự phát triển nhanh nhất của chúng:

$$f + g \in \max(\Theta(f(n)), \Theta(g(n))),$$

có thể được viết như sau:

$$f \in \Theta(g) \Rightarrow f + g \in \Theta(g)$$

hoặc là:

$$\Theta(c_1 f(n) + c_2 g(n)) \in \max(\Theta(f(n)), \Theta(g(n))), \text{ với } c_1, c_2 > 0.$$

6. Nếu $f(n)$ là đa thức bậc chính xác d thì $f \in \Theta(n^d)$.

Định nghĩa 1.31. Một quan hệ đồng thời có các thuộc tính phản xạ, đối xứng và tính nhanh là *quan hệ tương đương*.

Chúng ta hãy xác định quan hệ $R = “\text{thuộc } \Theta(\dots)”$ được cho dưới dạng $(f, g) \in R$ nếu và chỉ khi $f \in \Theta(g)$. Từ các tính chất của $\Theta(\dots)$ và từ định nghĩa 1.3.2. theo đó R là một quan hệ tương đương.

Chứng minh các tính chất trên theo định nghĩa:

Để khẳng định rằng hàm số f thuộc $\Theta(g)$, cần và đủ để tìm một số tự nhiên n_0 và hai hằng số dương c_1 và c_2 sao cho với mọi số tự nhiên $n (n > n_0)$ thì nó có giá trị là bất đẳng thức $c_1.g(n) \leq f(n) \leq c_2.g(n)$.

Ví dụ 1.9. Lấy ví dụ hàm $\frac{n^2}{2} - 3n$. Ta sẽ chứng tỏ rằng nó thuộc tập hàm $\Theta(n^2)$. Ta tìm các hằng số c_1, c_2 và n_0 sao cho $c_1.n^2 \leq n^2/2 - 3n \leq c_2.n^2 (n > n_0)$. Chia cho n^2 (với ràng buộc bổ sung $n > 0$) ta được: $c_1 \leq 1/2 - 3/n \leq c_2$. Một lựa chọn có thể là $c_1 = 0, c_2 = 1/2$

và $n_0 = 5$. Chúng ta để độc giả thực hiện kiểm tra tương ứng. Chúng ta sẽ lưu ý rằng các giá trị c_1, c_2 và n_0 này không có nghĩa là duy nhất. Ví dụ, chúng ta có thể chọn $c_1 = 1/20, c_2 = 1$ và $n_0 = 100$. Chúng phụ thuộc nhiều vào các hệ số của đa thức đã xét và đối với đa thức khác trong trường hợp chung chúng sẽ khác nhau đáng kể.

Ví dụ 1.10. Xét hàm số $2n^3$. Ta sẽ chứng tỏ rằng nó không thuộc tập các hàm $\Theta(n^2)$. Thật vậy, nếu giả sử ngược lại thì phải tồn tại một hằng số c_2 sao cho $6n^3 \leq c_2 \cdot n^2$. Nhưng nếu chúng ta chia cho n^2 (với giới hạn bổ sung $n > 0$) hai vế của bất đẳng thức, chúng ta nhận được bất đẳng thức $6n \leq c_2$, và do đó: $n \leq c_2/6$. Vì vậy, nó chỉ ra rằng n nhỏ hơn một số hằng số. Rõ ràng, không thể tìm thấy n sao cho với mọi $n, n > n_0, n \leq c_2/6$ thỏa mãn: bên trái của bất đẳng thức ta có số không giới hạn ở đỉnh và bên phải là hằng số.

Tính chất tiệm cận:

1. $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \in \mathbb{R}^* \Rightarrow f(n) \in \Theta(g(n)), g(n) \in \Theta(f(n))$.
2. $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in O(g(n)), g(n) \notin \Theta(f(n)),$ nghĩa là $f(n) \in o(g(n))$.
3. $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) \in \Omega(g(n)), f(n) \notin \Theta(g(n)),$ nghĩa là $f(n) \in \omega(g(n))$.

Các thuộc tính này có thể được sử dụng như một cách thay thế để xác minh rằng một hàm thuộc về một lớp hàm cụ thể.

Ví dụ 1.11. Đánh giá khẳng định.

$$10n \in \Theta(n); \quad 4n^2 - 5n + 2 \in \Theta(n^2);$$

$$4n^2 - 5n + 2 \in \Theta(6n^2 - n + 1); \quad \log_{10} n \in \Theta(\log_2 n).$$

Ví dụ 1.12. Đánh giá phủ định.

$$10n \notin \Theta(1); \quad 10n \notin \Theta(n^2);$$

$$4n^2 - 5n + 2 \notin \Theta(n^3); \quad 4n^2 - 5n + 2 \notin \Theta(n + 1).$$

Định nghĩa 1.32. Độ phức tạp $\Theta(1)$ được gọi là *hằng số*, $\Theta(\log n)$ - *logarit*, $\Theta(n)$ - *tuyến tính*, $\Theta(n^2)$ - *bậc hai*, $\Theta(c^n)$ - *hàm mũ*. Khi hàm f là một đa thức thì độ phức tạp $\Theta(f)$ được gọi là *đa thức*.

Bài tập

- 1.117. Kiểm tra theo định nghĩa rằng $5n^3 - 5n^2 + 5 \in \Theta(n^3)$.
- 1.118. Chứng minh rằng $\sqrt{n} \notin \Theta(n^{0.48})$.
- 1.119. Chứng minh các tính chất tổng quát trên của $\Theta(F)$.
- 1.120. Chứng minh tính chất tiệm cận trên của $\Theta(F)$.
- 1.121. Kiểm tra xem quan hệ $R = \text{"có thuộc } \Theta(\dots)}$ (được cho dưới dạng $(f, g) \in R$ nếu và chỉ khi $f \in \Theta(g)$) là quan hệ tương đương.
- 1.122. Chỉ ra các khẳng định đúng:
- $4n^3 + 5n^2 + 5 \in \Omega(n^4)$;
 - $\log_2 n \notin \Omega(\sqrt{n})$;
 - $4n^3 + 5n^2 + 5 \in \Omega(7n^4 - 10)$;
 - $n \in \Omega(\sqrt{n})$;
 - $\log_2 n \in O(\sqrt{n})$;
 - $10n + 3 \in O(n)$;
 - $10n \in O(3n^4 - 10n^2 + 7)$;
 - $5n + 1 \in O(\sqrt{n})$.
- 1.123. Chỉ ra các khẳng định đúng:
- $\Theta(F) = O(F) \cap \Omega(F)$;
 - $\omega(F) = \Omega(F) \setminus \Theta(F)$;
 - $o(F) = O(F) \setminus \Theta(F)$;
 - $O(c_1 f + c_2 g) = O(\max(f, g))$;
 - $f \in \Omega(g) \Rightarrow g \in \Theta(f)$;
 - $f \in O(g) \Rightarrow g \in \Omega(f)$;
 - $f \in \Omega(g) \Leftrightarrow g \notin O(f)$.
- 1.124. Chỉ ra các khẳng định đúng:
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+ \Rightarrow f(n) \in O(g(n)), g(n) \in O(f(n))$;
 - $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in O(g(n)), g(n) \notin O(f(n))$;
 - $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow f(n) \notin O(g(n)), g(n) \in O(f(n))$;
 - $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+ \Rightarrow f(n) \in \Theta(g(n))$;

e) $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) \in \Omega(g(n)), g(n) \notin \Theta(f(n)).$

1.13.5. Hàm tiệm cận và số thực

Các quan hệ giữa các hàm tiệm cận được thảo luận ở trên có các tính chất gần với các quan hệ cổ điển trên các số thực. Các liên kết được cho trong [Bảng 1.6](#).

Tuy nhiên, mỗi quan hệ chỉ là một phần, vì các hàm tiệm cận không có một tính chất quan trọng của số thực: tam phân. Nghĩa là, với mọi số thực a và b có đúng một trong các quan hệ hợp lệ: $a < b, a = b$ và $a > b$. Ví dụ, đối với các hàm n và $n^{1+\cos n}$ không có quan hệ nào trong số các quan hệ trên là hợp lệ. Chúng ta để lại chứng minh của quan hệ sau cho người đọc.

quan hệ tiệm cận	quan hệ số thực
$f \in O(g)$	$a \leq b$
$f \in \Omega(g)$	$a \geq b$
$f \in \Theta(g)$	$a = b$
$f \in o(g)$	$a < b$
$f \in \omega(d)$	$a > b$

Bảng 1.6. Mỗi liên hệ giữa quan hệ về tiệm cận của hàm số và số thực.

Bài tập

- ▷ **1.125.** Để đưa ra một ví dụ khác về một cặp hàm số tiệm cận không so sánh được.
- ▷ **1.126.** Một trong hai hàm $f(n)$ và $g(n)$ có cần thiết là hợp của hai hàm, một trong hai hàm đó phải tuần hoàn, để f và g có tiệm cận không?

Tăng các hàm cơ bản

Các hàm sau đây thường được sử dụng nhất khi ước lượng độ phức tạp của các thuật toán: $c, \log n, n, n \log n, n^2, n^3, n^k, 2^n, n!, n^n$. Ở

đây chúng ta đã sắp xếp chúng để tăng trưởng nhanh hơn. Để người đọc hiểu rõ hơn về tốc độ tăng trưởng của chúng, chúng ta áp dụng Bảng 1.7, Hiển thị các giá trị của các hàm ở các giá trị khác nhau của đối số n của chúng.

Trực tiếp từ Bảng 1.7 người ta thấy rằng:

- Hàm lũy thừa phát triển nhanh hơn hàm lũy thừa:
 $n^k \in O(b^n)$, với mỗi $b > 1, k \geq 0, n$ - tự nhiên.
- Hàm số logarit tăng chậm hơn lũy thừa:
 $\log_b n \in O(n^k)$, với mỗi $b > 1, k > 0, n$ - tự nhiên.

Hàm số	Giá trị				
	$n = 1$	$n = 2$	$n = 10$	$n = 100$	$n = 1000$
5	5	5	5	5	5
\log_n	0	1	3,32	6,64	9,96
n	1	2	10	100	1000
$n \log n$	0	2	33,2	664	9966
n^2	1	4	100	10000	10^6
n^3	1	8	1000	10^6	10^9
2^n	2	4	1024	10^{30}	10^{300}
$n!$	1	2	3628800	10^{157}	10^{2567}
n^n	1	4	10^{10}	10^{200}	10^{3000}

Bảng 1.7. Tăng một số hàm tiệm cận được sử dụng phổ biến hơn.

Bài tập

- 1.127. Chứng minh rằng $n!$ phát triển nhanh hơn c^n , với mọi $c > 0$.
- 1.128. Sắp xếp hàm $c^n, c > 0$ theo thứ tự tăng dần của các hàm thường dùng nhất (xem Bảng 1.7).

1.13.6. Xác định độ phức tạp của một thuật toán

Chúng ta sẽ xem xét một số thuộc tính cơ bản, với sự trợ giúp của chúng ta sẽ có thể xác định độ phức tạp của một thuật toán đối

với một triển khai C nhất định. Chúng ta sẽ sử dụng $T(\text{mã})$ để chỉ ra mức độ phức tạp của cả một hoạt động riêng lẻ và một đoạn chương trình. Đôi khi, khi mã được ngụ ý, chúng ta sẽ sử dụng ký hiệu $T(n)$ để chỉ ra rõ ràng rằng độ phức tạp là một hàm của biến n . Chúng ta sẽ làm việc độc quyền với ký hiệu $O(\dots)$. Chúng ta để người đọc tìm ra vị trí có thể thay thế $O(\dots)$ bằng ước lượng chính xác hơn $\Theta(\dots)$.

Phép tính cơ bản

Độ phức tạp của một phép toán cơ bản là một hằng số, tức là $O(1)$. Không dễ để định nghĩa một phép toán cơ bản là gì. Trong các trường hợp khác nhau, chúng ta sẽ cho phép mình thay đổi định nghĩa. Về nguyên tắc, hoạt động cơ bản là hoạt động được thực hiện trong một thời gian không đổi, bất kể số lượng dữ liệu được xử lý. Các phép toán cơ bản trong trường hợp chung là, ví dụ, chiếm đoạt, cộng, nhân, v.v. Tuy nhiên, khi làm việc với các số có 100 chữ số, thật khó chấp nhận phép nhân như một phép toán cơ bản. Sẽ là không tốt nếu sử dụng các hàm lượng giác ($\sin, \cos, \text{v.v.}$), số mũ, logarit và các hàm thư viện khác trong C, được tính theo hàng trong C là không tốt cho các phép toán cơ bản. Việc gọi một hàm như vậy gây ra một chu kỳ để tính giá trị cần thiết. Mặt khác, các hàng này được tính toán cho đến khi đạt được độ chính xác nhất định, và số lần lặp để tính giá trị cần thiết có thể được coi là độc lập với n . Điều này có ảnh hưởng đến việc đánh giá độ phức tạp của thuật toán hay không và như thế nào thì tùy thuộc vào từng trường hợp và cần được xem xét cụ thể.

Trình tự các toán tử

Độ phức tạp về thời gian của một chuỗi các toán tử được xác định bởi một toán tử chậm hơn. Nếu toán tử s_1 với độ phức tạp F_1 được sau bởi toán tử s_2 với độ phức tạp F_2 , chúng ta có thể viết:

$$T(s_1) \in O(F_1), T(s_2) \in O(F_2) \Rightarrow T(s_1; s_2) \in O(\max(F_1, F_2))$$

Điều này tương đương với quy tắc:

$$f_1 + f_2 \in (\max(f_1, f_2))$$

Kết hợp các toán tử

Khi một toán tử được bao gồm trong phạm vi của một toán tử khác, độ phức tạp được tính như một tích của độ phức tạp của

chúng, tức là

$$T(s_1) \in O(F_1), T(s_2) \in O(F_2) \Rightarrow T(s_1\{s_2\}) \in O(F_1.F_2))$$

Điều này tương đương với quy tắc:

$$f_1.f_2 \in O(f_1.f_2)$$

Cấu trúc mệnh đề if

```
if (p)
  s1;
else
  s2;
```

Nếu độ phức tạp của p, s_1 và s_2 là $O(P), O(F_1), O(F_2)$, thì độ phức tạp của đoạn được hiển thị là tối đa ($O(P), O(F_1), O_2)$), tức là độ phức tạp của hàm phát triển nhanh nhất giữa P, F_1 và F_2 . Ở đây chúng ta giả sử rằng điều kiện p không phải là hằng số Boolean, tức là tùy thuộc vào đầu vào, nó có thể vừa đúng vừa sai.

$$T(p) \in O(P), T(s_1) \in O(F_1), T(s_2) \in O(F_2)$$

$$\Rightarrow T(\text{if } (p) \text{ } s_1; \text{ else } s_2) \in \max(O(P), O(F_1), O(F_2))$$

Chúng ta sẽ cố gắng chứng minh tài sản này. Giả sử rằng điều kiện p là đúng. Khi đó *cấu trúc if* tương đương với dãy $p; s_1$ (Lưu ý rằng điều kiện đã được kiểm tra.). Theo quy tắc của trình tự, độ phức tạp trong trường hợp này là $\max(O(P), O(F_1))$. Theo cách tương tự, chúng ta hiểu rằng độ phức tạp trong trường hợp p sai là cực đại ($O(P), O(F_2)$). Vì không biết trước chuỗi nào trong hai chuỗi sẽ được thực thi, chúng ta có thể hạn chế mức độ phức tạp của *cấu trúc if* từ phía trên, giả sử trường hợp nghiêm trọng hơn. Đây là cách chúng ta nhận được:

$$\begin{aligned} T(\text{if } (p) \text{ } s_1; \text{ else } s_2) &\in \max(\max(O(P), O(F_1)), \max(O(P), O(F_2))) \\ &= \max(O(P)), O(F_1), O(F_2)) \end{aligned}$$

Theo cách tương tự, sự phức tạp của các cấu trúc chuyển mạch được suy ra.

Chu trình lặp

Hãy nhìn vào chu kỳ:

```

fact = 1;
for (i = 1; i <= n; i++)
    fact *= i;

```

Chúng ta có thể giả sử rằng phần thân của chu trình mất một khoảng thời gian c không đổi, không phụ thuộc vào n . Độ phức tạp của toán tử vòng lặp *for* là $O(n)$. Sau đó, theo quy tắc thành phần về độ phức tạp của toàn bộ chu trình, chúng ta thu được $O(c.n)$, tức là $O(n)$. Ở đây chúng ta phải thêm độ phức tạp của lần khởi tạo ban đầu trước chu trình (có độ phức tạp $O(1)$), trong đó, theo quy tắc trình tự, chúng ta nhận được: $O(1 + n)$. Cuối cùng, độ phức tạp hóa ra là $O(n)$.

Chu trình lồng nhau

```

sum = 0;
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        sum++;

```

Có thể dễ dàng suy ra độ phức tạp của hai hoặc nhiều chu trình lồng nhau với các bộ đếm độc lập lẫn nhau. Trong trường hợp hai chu trình lồng nhau từ đoạn trên, nó là $f \in n.O(g)$, trong đó g là độ phức tạp của chu trình bên trong. Nhưng $g \in O(n)$, thì $f \in O(n.n)$, tức là $f \in O(n^2)$.

Ở đây và bên dưới, chúng ta thường khởi tạo một tổng biến đặc biệt bằng 0, và sau đó chúng ta sẽ sử dụng *sum++* làm toán tử bên trong nhất trong các vòng lặp. Điều này cho phép người đọc tò mò kiểm tra lý thuyết lý thuyết của chúng ta trong thực tế. Với mục đích này, chỉ cần quan sát giá trị của *sum* sau khi thực hiện phân mảnh đối với các giá trị khác nhau của n là đủ.

```

sum = 0;
for (i = 0; i < n-1; i++)
    for (j = i+1; j < n; j++)
        sum++;

```

Trong ví dụ này, trong bước đầu tiên của chu kỳ ngoài, $i = 0$, chu kỳ bên trong sẽ được thực hiện $n - 1$ lần. Trong bước thứ hai, đối với $i = 1$, chu trình bên trong sẽ được thực hiện $n - 2$ lần, sau

đó là $n - 3$ lần, v.v., và cuối cùng, đối với $i = n - 2$, nó sẽ chỉ được thực hiện một lần. Ta có một cấp số cộng với phần tử đầu tiên 1, phần tử cuối cùng $n - 1$ và bước 1. Như vậy sum`++` sẽ được thực hiện $\frac{n(n - 1)}{2}$ lần, tức là độ phức tạp của phân mảnh sẽ là $O(n^2)$. Chúng ta giả sử rằng việc thực thi sum `++` có độ phức tạp không đổi $O(1)$.

Ví dụ 1.13. Đoạn mã

```
sum = 0;
for (i = 0; i < n*n; i++)
    sum++;
```

Độ phức tạp là $O(n^2)$. (Tại sao?)

Ví dụ 1.14. Đoạn mã

```
sum = 0;
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        if (i == j)
            for (k = 0; k < n; k++)
                sum++;
```

Câu lệnh `if` sẽ được thực hiện n^2 lần, nhưng chỉ n lần kết quả kiểm tra `i == j` là đúng. Vì độ phức tạp của chu trình trong cùng là tuyến tính nên chúng ta thu được tổng độ phức tạp $O(n^2)$. Tất nhiên, ước lượng $O(n^3)$ cũng đúng, nhưng nó không chính xác hơn. Độ phức tạp ở đây là $\Theta(n^2)$.

Ví dụ 1.15. Đoạn mã

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        if (i == j)
            break;
```

Làm thế nào nó có thể được chỉ ra rằng độ phức tạp trong đoạn trên một lần nữa là $O(n^2)$ chứ không phải ít hơn? Lưu ý rằng, không giống như ví dụ trước, không có chu trình tuyến tính bổ sung!

Ví dụ 1.16. Đoạn mã

```
sum = 0;
for (i = 1; i <= n; i++)
    for (j = 1; j <= i*i; j++)
        sum++;
```

Chu trình bên trong sẽ được thực hiện n lần. Từ công thức

$$\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6}$$

chúng ta nhận được rằng độ phức tạp của phân mảnh là $O(n^3)$. Trên thực tế, chúng ta có thể quyết định rằng độ phức tạp của chu trình ngoài là $O(n)$ và độ phức tạp của chu trình trong là $O(n^2)$, bởi vì giá trị lớn nhất của i là n . Do đó với $i == n$ chúng ta có $i * i == n^2$ và một lần nữa chúng ta đạt đến độ phức tạp $O(n^3)$.

Ví dụ 1.17. Đoạn mã

```
sum = 0;
for (i = 0; i < n; i++)
    for (j = 0; j < i*i; j++)
        for (k = 0; k < j*j; k++)
            sum++;
```

Một lần nữa, có thể tính toán độ phức tạp bằng cách phân tích số lượng hoạt động trong chu kỳ trong cùng. Tuy nhiên, trong thực tế, có thể đạt được kết quả tương tự và dễ dàng hơn nhiều, chỉ sử dụng các thuộc tính cơ bản của ký hiệu $O(\dots)$: Trong ví dụ trước, chúng ta đã tính toán độ phức tạp của hai chu trình lồng nhau bên ngoài - $O(n^3)$. Bản thân chu trình cuối cùng có độ phức tạp $O(n^4)$ - điều này dễ thấy nếu chúng ta xét trường hợp $i = n$, thì giới hạn trên của k là $j^2 = i^4 = n^4$. (Độ phức tạp tổng thể của phân mảnh là gì và tại sao?)

Độ phức tạp logarit

Hãy xem phân đoạn chương trình:

```
cho (sum = 0, h = 1; h <n; h * = 2)
tổng ++;
```

Ở đây h nhận các giá trị $1, 2, 4, \dots, 2^k, \dots$ cho đến khi nó đạt đến n . Do đó, sum `++` được thực hiện $\lfloor \log_2 n \rfloor$ lần và độ phức tạp của thuật toán là $O(\log_2 n)$. Lưu ý rằng độ phức tạp được đặt bởi một logarit, là một hàm thực. Như đã lưu ý ở trên, đây không phải là vấn đề và chúng ta sẽ hiểu được xấp xỉ / ràng buộc số nguyên thích hợp.

Tìm kiếm nhị phân có độ phức tạp thuật toán như vậy, trong đó ở mỗi bước, khoảng thời gian tìm kiếm được chia thành hai (gần như) phần bằng nhau. Chúng ta sẽ không đề cập đến thuật toán này, vì chúng ta sẽ xem xét nó chi tiết hơn trong phần sau của cuốn sách (xem 4.3.).

Dưới đây, thay vì $O(\log_2 n)$, chúng ta sẽ viết $O(\log n)$. Một mặt, bỏ cơ số ở lôgarit nhị phân là một quy ước tiêu chuẩn. Mặt khác, nó không quan trọng trong ký hiệu tiệm cận do tính chất (4) của 1.1.1. độ, logarit, căn bậc n . Nếu chúng ta thích làm việc với cơ sở khác c ($c > 0, c \neq 1$) thì không có gì thay đổi, bởi vì $\log_2 x = \log - cx / \log_c 2$, nhưng $\log_c 2$ là một hằng số và bị bỏ qua trong ký hiệu tiệm cận. Như vậy $O(\log_2 n) \equiv O(\log_c n)$.

Thuật toán đệ quy

Phân tích đệ quy trong trường hợp tổng quát là không hề nhỏ. Thông thường, độ phức tạp của thuật toán phụ thuộc vào kiểu $T(n) = f(T(n - 1))$. Để tìm ra loại phức tạp rõ ràng, cần phải giải quyết sự phụ thuộc thường xuyên, nói chung là khó. May mắn thay, trong hầu hết các trường hợp thực tế thú vị, điều này không quá khó và đôi khi có thể được thực hiện bằng các phương tiện khác.

Phép tính gai thừa

Hãy xem xét chức năng:

```
unsigned fact(unsigned n)
{ if (n < 2)
    return 1;
  return n * fact(n-1);
}
```

Trong trường hợp này, đệ quy tương đương với một vòng lặp for duy nhất, từ đó chúng ta có thể dễ dàng thu được $O(n)$ về độ phức tạp.

Tính số Fibonacci

Tuy nhiên, mọi thứ không phải lúc nào cũng đơn giản như vậy. Lấy ví dụ, các số Fibonacci mà chúng ta đã hiển thị trong 1.2.2 có thể được tìm thấy cực kỳ kém hiệu quả với đệ quy:

```
unsigned fib(unsigned n)
{
    if (n < 2)
        return 1;
    return fib(n-1) + fib(n-2);
}
```

Tại $n = 0$ và $n = 1$, chúng ta có độ phức tạp về thời gian không đổi: một lần kiểm tra cơ bản và trả về một kết quả. Trong trường hợp khác, chúng ta có kiểm tra lại tương tự, nhưng lần này được theo sau bởi hai tham chiếu đệ quy. Nói chung, các công thức hợp lệ:

$$\begin{aligned} T(0) &= T(1) = O(1) \\ T(n) &= T(n - 1) + T(n - 2) + O(1), \quad n \geq 2 \end{aligned}$$

Các phụ thuộc trên rất giống với định nghĩa của số Fibonacci:

$$\begin{aligned} f_0 &= f_1 = 1 \\ f_n &= f_{n-1} + f_{n-2} \end{aligned}$$

Ngay sau đó $T(n) \geq f_n$. Tuy nhiên, đối với số Fibonacci, bất đẳng thức có giá trị: $\left(\frac{3}{2}\right)^{n-1} \leq f_n \leq 2^n, n \geq 1$ (Chứng minh một chút bằng quy nạp.). Do đó, nó chỉ ra rằng $T(n)$ phát triển theo cấp số nhân.

Tuy nhiên, nếu chúng ta đùa tháo vát để không tính toán lại một số thứ mà chúng ta đã xem xét, chúng ta có thể giảm độ phức tạp của thuật toán xuống $O(n)$:

```
unsigned long f[MAX] = {0,0,0,...};
unsigned long fib(unsigned n)
{
    if (0 == f[n])
        if (n < 2)
            f[n] = 1;
```

```

    else
        f[n] = fib(n-1) + fib(n-2);
    return f[n];
}

```

Bài tập

- ▷ 1.129. Nhớ lại rằng đối với việc xây dựng **if-then-else** chúng ta đã có

$$\begin{aligned} T(p) &\in O(P), T(s_1) \in O(F_1), T(s_2) \in O(F_2) \\ \Rightarrow T(\text{if } (p)s_1; \text{else } s_2) &\in \max(O(P), O(F_1), O(F_2)). \end{aligned}$$

Tìm một biểu thức thuộc loại này cho công tắc cấu trúc switch.

- ▷ 1.130. Chứng minh rằng độ phức tạp của đoạn chương trình từ Ví dụ 1.13 ở trên là $\Theta(n^2)$.

- ▷ 1.131. Chứng minh rằng độ phức tạp của đoạn chương trình từ Ví dụ 1.15 ở trên là $\Theta(n^2)$.

- ▷ 1.132. Để đưa ra ước tính kiểu $\Theta(\dots)$ cho độ phức tạp của đoạn chương trình từ Ví dụ 1.17.

- ▷ 1.133. Có đúng không khi thay thế $O(\dots)$ bằng $\Theta(\dots)$ trong suốt 1.4.4.?

- ▷ 1.134. Chứng minh rằng đối với các số Fibonacci thì đẳng thức tồn tại:

$$\left(\frac{3}{2}\right)^{n-1} \leq f_n \leq 2^n, n \geq 1$$

- ▷ 1.135. Xác định độ phức tạp $\Theta(\dots)$ của đoạn sau:

```

for (i = 0; i < 2*n; i++)
    for (j = 0; j < 2*n; j++)
        if (i < j) for (k = 0; k < 2*n; k++)
            break;

```

- ▷ 1.136. 7. Xác định độ phức tạp $\Theta(\dots)$ của đoạn sau:

```
unsigned sum = 0;
for (i = 0; i < n; i++)
    for (j = i+1; j < i*i; j++)
        sum++;
```

- ▷ 1.137. Xác định độ phức tạp $\Theta(\dots)$ của đoạn sau:

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        if (i==j)
            for (k = 0; k < n; k++)
                break;
```

- ▷ 1.138. Xác định độ phức tạp $\Theta(\dots)$ của đoạn sau:

```
unsigned sum = 0;
for (i = 0; i < n; i++)
    for (j = 0; j < i*i; j++)
        sum++;
```

- ▷ 1.139. Xác định độ phức tạp $\Theta(\dots)$ của hàm trib():

```
unsigned trib(unsigned n)
{
    if (n < 3)
        return 1;
    if ((n % 2) == 1)
        return trib(n-1) + trib(n-2) + trib(n-3);
    else
        return trib(n / 3) + trib(n / 2);
}
```

- ▷ 1.140. Xác định độ phức tạp $\Theta(\dots)$ của hàm streep():

```
unsigned fib(unsigned n)
{
    if (n < 2)
        return 1;
    return fib(n-1) + fib(n-2);
}
```

```
void streep(unsigned n) {
    fib(fib(n));
}
```

- 1.141. Xác định độ phức tạp $\Theta(\dots)$ của đoạn sau:

```
int n = 10;
int i;
for (i = 0; i < n; (n = i)++);
```

1.14. Phương trình đặc trưng

Hãy quay lại các số Fibonacci. Lần này chúng ta sẽ không tập trung vào sự phức tạp của việc thực hiện đệ quy ngày thơ của đoạn trước, mà là giải quyết sự phụ thuộc lặp lại được thiết lập bởi định nghĩa của số Fibonacci. Kết quả là, chúng ta có được công thức Moavr nổi tiếng. Nhưng trước đó - một lý thuyết nhỏ ...

1.14.1. Phương trình thuần nhất tuyến tính với nghiệm đơn giản

Trong quá trình phân tích độ phức tạp của các thuật toán máy tính, thường thu được các phụ thuộc thường xuyên của kiểu:

$$a_0 T(n) + a_1 T(n-1) + \cdots + a_k T(n-k) = 0 \quad (1.14)$$

Ở đây $T(n)$ là một hàm chưa biết. Để có thể ước tính độ phức tạp của thuật toán tiệm cận, chúng ta cần tìm toàn bộ hoặc ít nhất một phần dạng rõ ràng của $T(n)$, tức là để giải quyết sự phụ thuộc thường xuyên. Cách tiếp cận tiêu chuẩn là giảm vấn đề thành giải một phương trình thuần nhất tương ứng. Cho $T(n) = x^n$, trong đó ta nhận được:

$$a_0 x^n + a_1 x^{n-1} + \cdots + a_k x^{n-k} = 0 \quad (1.15)$$

Một nghiệm của phương trình trên, cũng như của bất kỳ phương trình thuần nhất nào, là $x = 0$, không được quan tâm. Để tìm các nghiệm còn lại, ta chia cho $x^n - k$ (với $x \neq 0$) và thu được đa thức bậc k - đa thức đặc trưng của sự phụ thuộc hồi quy ban đầu:

$$a_0 x^k + a_1 x^{k-1} + \cdots + a_k = 0 \quad (1.16)$$

Phương trình kết quả là một đa thức bậc k. Theo định lý cơ bản của đại số, nó có đúng k căn (không nhất thiết khác và không nhất thiết thực, tức là có thể phức). Hãy ký hiệu chúng bằng $\alpha_i (1 \leq i \leq k)$. Sau đó chúng ta có:

$$0 = a_0x^k + a_1x^{k-1} + \cdots + a_k = a_0(x - \alpha_1)(x - \alpha_2)\dots(x - \alpha_k) \quad (1.17)$$

Nếu chúng ta nhân phương trình (1.17) với x^{n-k} , chúng ta nhận được rằng các số $\alpha_i^n (1 \leq i \leq k)$ không chỉ là nghiệm của phương trình (1.16) mà còn của (1.15). Quay trở lại vị trí $T(n) = x^n$, ta thu được α_i^n là nghiệm nguyên của sự phụ thuộc hồi quy ban đầu (1.14).

Phương trình (1.14) thuận nhất và có vô số nghiệm. Nếu các số α_i khác nhau, thì tập hợp các số α_i^n biểu diễn một hệ nghiệm cơ bản của (1.14), tức là tất cả các nghiệm khác thu được dưới dạng kết hợp tuyến tính của α_i^n . Như vậy nghiệm tổng quát của (1.14) có dạng:

$$T(n) = c_1\alpha_1^n + c_2\alpha_2^n + \cdots + c_k\alpha_k^n \quad (1.18)$$

Ở đây các hệ số c_1, c_2, \dots, c_n là các hằng số được xác định duy nhất bởi các điều kiện biên. Dưới đây chúng ta sẽ xem cách này hoạt động như thế nào trong thực tế.

Ví dụ 1.18. Hãy xem xét các số Fibonacci thỏa mãn sự phụ thuộc lặp lại:

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

Phương trình đặc trưng tương ứng của loại (1.17) là:

$$x^2 = x + 1$$

và gốc rễ của nó là:

$$\alpha_{1,2} = \frac{1 \pm \sqrt{5}}{2}$$

Chúng ta đang tìm kiếm một giải pháp thuộc loại (1.18), tức là:

$$T(n) = c_1\alpha_1^n + c_2\alpha_2^n \quad (1.19)$$

Chúng ta xác định các hằng số c_1 và c_2 từ các điều kiện biên $T(0) = 0$ và $T(1) = 1$, thay thế vào (1.19), ở đó chúng ta nhận được

hệ thống:

$$\begin{aligned} T(0) &= 0 = c_1 + c_2 \\ T(1) &= 1 = c_1\alpha_1 + c_2\alpha_2 \end{aligned}$$

chúng ta rút ra:

$$c_1 = \frac{1}{\sqrt{5}}, c_2 = -\frac{1}{\sqrt{5}}$$

Bây giờ chúng ta thay thế trong (1.19) và thu được dạng rõ ràng của $T(n)$. Công thức kết quả được gọi là công thức Moaver.

$$T(n) = \frac{1}{\sqrt{5}} \left[\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right].$$

Công thức trên cho thấy rõ ràng rằng số Fibonacci tăng theo cấp số nhân khi tăng n . Thoạt nhìn, công thức có vẻ lạ - nó chứa các số vô tỉ. Chúng ta giao nó cho người đọc để đảm bảo rằng chúng được viết tắt và với mỗi n tự nhiên sẽ thu được các số Fibonacci tương ứng.

1.14.2. Phương trình thuận nhất tuyến tính với nhiều nghiệm

Trong trường hợp phương trình (1.17) có nhiều nghiệm, nghiệm tổng quát có dạng hơi khác so với (1.18). Nếu α là căn kép thì căn đó cũng sẽ là $n\alpha^n$. Lưu ý rằng gốc này không thể nhận được dưới dạng kết hợp tuyến tính của các gốc khác, tức là nó về cơ bản là mới và phải được đưa vào hệ thống cơ bản của các quyết định. Nếu căn bậc ba thì ngoài $n\alpha^n$ ta nên thêm $n^2\alpha^n$. Nói chung, trong trường hợp một căn bậc hai, các căn α sẽ là tất cả các đơn thức $n^s\alpha^n$, $0 \leq s \leq p$.

Đổi tên các gốc α_i ($1 \leq i \leq k$) để loại bỏ nhiều gốc. Gọi các nghiệm nguyên khác nhau (1.16) là $\beta_1, \beta_2, \dots, \beta_l$, $1 \leq l \leq k$. Gọi q_j là bội của nghiệm β_j , $1 \leq j \leq l$. Khi đó, nghiệm chung của (1.14) sẽ giống như sau:

$$T(n) = \sum_{j=1}^l \sum_{r=0}^{q_j-1} c_{j,r+1} n^r \beta_j^n. \quad (1.20)$$

Các số $c_{j,r+1}$ là các hệ số được xác định rõ ràng, như trên, theo giới hạn các điều kiện.

Ví dụ 1.19.

$$T(n) = \begin{cases} 3 & n = 0; 1 \\ 5 & n = 2 \\ 4T(n-1) - 5T(n-2) + 2T(n-3) & n \geq 2 \end{cases}$$

Phương trình đặc tính tương ứng có dạng:

$$x^3 = 4x^2 - 5x + 2$$

Hoặc, nếu chúng ta chuyển mọi thứ sang bên trái:

$$x^3 - 4x^2 + 5x - 2 = 0$$

Giải quyết nó, chúng ta nhận được:

$$x_1 = x_2 = 1, x_3 = 2$$

Chúng ta tìm kiếm dạng rõ ràng của $T(n)$ ở dạng (1.20):

$$T(n) = c_{1.1} \cdot 1^n + c_{1.2} \cdot n \cdot 1^n + c_{2.1} \cdot 2^n \quad (1.21)$$

Chúng ta xác định các hằng số $c_{1.1}, c_{1.2}$ và $c_{2.1}$ từ các điều kiện biên cho $n = 0, 1$ và 2 . Chúng ta nhận được hệ thống:

$$\begin{aligned} T(0) &= 3 = c_{1.1} + c_{2.1} \\ T(1) &= 3 = c_{1.1} + c_{1.2} + 2c_{2.1} \\ T(2) &= 5 = c_{1.1} + 2c_{1.2} + 4c_{2.1} \end{aligned}$$

ở đâu:

$$c_{1.1} = 1, c_{1.2} = -2, c_{2.1} = 2$$

Bây giờ chúng ta thay thế trong (1.21) và nhận được:

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

1.14.3. Phương trình tuyến tính không thuần nhất

Trong quá trình phân tích thuật toán thường phải phân tích các phương trình không thuần nhất. Hãy xem xét nhiệm vụ sau:

Bài toán: Hàm của C được đưa ra:

```
float P(unsigned i, unsigned j)
{ if (0 == i)
    return 1.0;
else if (0 == j)
    return 0.0;
else
    return p * P(i - 1, j) + (1 - p) * P(i, j - 1);
}
```

Chúng ta muốn ước tính độ phức tạp về thời gian của phép nghịch đảo $P(n, n)$ dưới dạng một hàm của n .

Lời giải:

Chúng ta sẽ giải quyết vấn đề tổng quát hơn về ước lượng độ phức tạp của hàm đối với các tham số i và j , và sau đó chúng ta sẽ xem điều gì xảy ra với $i = j = n$.

Dễ dàng thấy rằng hàm $P()$ ở trên tính giá trị của hàm $P(i, j)$ cho bởi các phương trình truy hồi:

$$P(0, j) = 1, j = 1, 2, \dots, n$$

$$P(i, 0) = 0, i = 1, 2, \dots, n$$

$$P(i, j) = p.P(i - 1, j) + (1 - p).P(i, j - 1), i > 0, j > 0$$

Ở đây độ phức tạp phụ thuộc vào hai tham số nên khó tính toán, tuy nhiên có thể dễ dàng thấy rằng mọi thứ đối xứng với i và j . Do đó, giả sử $k = i + j$, chúng ta thu được:

$$T(1) = c$$

$$T(k) = 2T(k - 1) + d, k > 1$$

Lưu ý rằng ở đây chúng ta không cố gắng tìm dạng rõ ràng của hàm $P(i, j)$, mà để tìm ước lượng tiệm cận của việc thực hiện chương trình của hàm $P()$. Điều này cho chúng ta quyền không đi sâu quá giá trị của các hằng số c và d , trừ khi nó thực sự cần thiết. Chúng ta

sẽ thấy bên dưới rằng trong trường hợp của chúng ta, các *giá trị cụ thể* của chúng không liên quan.

Trước tiên, chúng ta hãy xem xét một cách đơn giản để có được phương trình đặc trưng trong trường hợp của chúng ta, trong đó một hằng số có liên quan đến quan hệ truy hồi. Chúng ta ghi lại sự phụ thuộc lặp lại cho hai phần tử liên tiếp của chuỗi:

$$\begin{aligned}T(k) &= 2T(k-1) + d \\T(k+1) &= 2T(k) + d\end{aligned}$$

Chúng ta lấy chúng ra theo nhóm, trong đó hằng số được giảm xuống, và chúng ta nhận được:

$$T(k+1) - T(k) = 2.[T(k) - T(k-1)]$$

tương đương với:

$$T(k+1) - 3T(k) + 2T(k-1) = 0.$$

Phương trình đặc trưng tương ứng là:

$$x^2 - 3x + 2 = 0$$

hoặc là

$$(x-1)(x-2) = 0,$$

trong đó người ta thấy trực tiếp nghiệm của phương trình là $\alpha_1 = 2$ và $\alpha_2 = 1$. Tức là chúng ta tìm kiếm dạng rõ ràng của $T(k)$ trong biểu mẫu:

$$T(k) = c_1\alpha_1^k + c_2\alpha_2^k$$

Hoặc sau khi thay thế:

$$T(k) = c_12^k + c_2$$

Các hệ số c_1 và c_2 có thể được tìm thấy bằng cách sử dụng các giá trị của $T(1)$ và $T(2)$. Chúng ta nhận được hệ thống:

$$\begin{aligned}T(1) &= c = 2c_1 + c_2 \\T(2) &= 2c + d = 4c_1 + c_2\end{aligned}$$

ở đâu: $c_1 = (c + d)/2; c_2 = -d$.

Từ $c_1 \neq 0$ lập tức có $T(k) \in \Theta(2^k)$. (Tại sao?) Bây giờ chúng ta hãy quay lại ứng dụng $k = i + j$, nơi chúng ta nhận được $\Theta(2^{i+j})$. Chúng ta hãy nhớ lại rằng chúng ta muốn ước tính độ phức tạp của đoạn chương trình trong địa chỉ $P(n, n)$, tức là cho $i = j = n$. Do đó ta nhận được $\Theta(2^{2n})$ hay cuối cùng là: $\Theta(4^n)$.

Định lý 1.9. Xét phương trình không thuần nhất:

$$a_0 T(n) + a_1 T(n-1) + \cdots + a_k T(n-k) = b_1 np_1(n) + b_2 np_2(n) + \cdots + b_s n p_s(n) \quad (1.22)$$

Ở đây b_1, b_2, \dots, b_s là các hằng số thực khác nhau và $p_1(n), p_2(n), \dots, p_s(n)$ là các đa thức của n . Gọi lũy thừa của các đa thức lần lượt là d_1, d_2, \dots, d_s . Khi đó phương trình đặc trưng của sự phụ thuộc hồi quy (1.22) có dạng:

$$(a_0 x^k + a_1 x^{k-1} + \cdots + a_k)(x - b_1)^{d_1+1}(x - b_2)^{d_2+1} \cdots (x - b_s)^{d_s+1} = 0 \quad (1.23)$$

Phương trình kết quả (1.23) được tiếp tục giải như trong trường hợp thuần nhất.

Ví dụ 1.20. Xem xét sự phụ thuộc thường xuyên:

$$T(n) = \begin{cases} 0 & n = 0 \\ 2T(n-1) + n + 2^n & n \geq 1 \end{cases}$$

Rõ ràng đây là sự phụ thuộc lặp lại không đồng nhất:

$$T(n) - 2T(n-1) = n + 2^n, \quad (1.24)$$

như $b_1 = 1, b_2 = 2, p_1(n) = n, p_2(n) = 1$, và do đó: $d_1 = 1, d_2 = 0$. Khi đó phương trình đặc trưng tương ứng có dạng:

$$(x - 2)(x - 1)^2(x - 2) = 0.$$

Nghiệm của phương trình là $x_1 = x_2 = 1, x_3 = x_4 = 2$. Chúng ta tìm dạng tương minh của $T(n)$:

$$T(n) = c_{1.1} \cdot 1^n + c_{1.2} \cdot n \cdot 1^n + c_{2.1} \cdot 2^n + c_{2.2} \cdot n \cdot 2^n \quad (1.25)$$

Để tìm các hằng số, lúc này điều kiện biên sẽ không đủ. Cần tính giá trị của $T(n)$ cho ba giá trị tiếp theo. Đây là cách chúng ta có được hệ thống:

$$\begin{aligned}T(0) &= 0 = c_{1,1} + c_{2,1} \\T(1) &= 3 = c_{1,1} + c_{1,2} + 2c_{2,1} + 2c_{2,2} \\T(2) &= 12 = c_{1,1} + 2c_{1,2} + 4c_{2,1} + 8c_{2,2} \\T(3) &= 35 = c_{1,1} + 3c_{1,2} + 8c_{2,1} + 24c_{2,2}\end{aligned}$$

Chúng ta giải nó và nhận được $c_{1,1} = -2, c_{1,2} = -1, c_{2,1} = 2$ và $c_{2,2} = 1$. Bây giờ chúng ta thay thế vào (1.25) và cho $T(n)$ chúng ta cuối cùng đến:

$$T(n) = -2 - n + 2^{n+1} + n \cdot 2^n. \quad (1.26)$$

Hệ thống trên có bốn ẩn số, khiến nó khó giải quyết (mặc dù không quá khó). Chúng ta có thể tìm các hệ số theo một cách khác - bằng cách thay (1.25) cho (1.24), chúng ta nhận được:

$$c_{2,2}2^n - c_{1,2}n + (2c_{1,2} - c_{1,1}) = 2^n + n. \quad (1.27)$$

Cân bằng các hệ số trước 2^n ở cả hai vế của đẳng thức, ta được $c_{2,2} = 1$. Tương tự, cân bằng các hệ số trước n cho ta $c_{1,2} = -1$. Bây giờ, biết $c_{1,2}$, từ phương trình của các số hạng tự do, chúng ta nhận được $c_{1,1} = -2$. Hằng số $c_{1,2}$ không thể đạt được theo cách này, nhưng nó xảy ra trực tiếp nếu chúng ta sử dụng điều kiện biên.

Giả sử rằng phương trình của ví dụ cuối cùng thu được là kết quả của việc phân tích một số thuật toán. Điều chúng ta quan tâm trong trường hợp này không phải là dạng rõ ràng của $T(n)$ như là một ước lượng tiệm cận của nó. Tất nhiên, khi biết dạng tường minh (1.26) của $T(n)$, chúng ta ngay lập tức nhận được rằng $T(n) \in \Theta(n2^n)$. Tuy nhiên, chúng ta có thể thiết lập điều này với ít nỗ lực hơn. Thật vậy, rõ ràng từ (1.25) rằng $T(n) \in O(n2^n)$. Nếu chúng ta hài lòng với kết quả này, chúng ta có thể dừng lại ở đó. Tuy nhiên, nếu chúng ta muốn có được một ước lượng có dạng $\Theta(\dots)$, thì điều này là không đủ, vì hệ số $c_{2,2}$ ở phía trước của hàm phát triển nhanh nhất $n2^n$ có thể là 0. Nhớ lại rằng từ (1.27) nó trực tiếp theo đó $c_{2,2} = 1$, tức là $c_{2,2} \neq 0$ và do đó ta kết luận rằng $T(n) \in \Theta(n2^n)$.

Nếu $c_{2,2}$ bằng 0, chúng ta sẽ kiểm tra hệ số $c_{2,1}$ trước hàm phát triển nhanh nhất tiếp theo 2^n , v.v.

Cân lưu ý hai điều:

1) *Không nhất thiết phải tìm tất cả các hệ số*. Trong trường hợp của chúng ta, nó đủ để chỉ ra rằng $c_{2,2} \neq 0$.

2) *Các điều kiện biên không nhất thiết phải phù hợp*. Chưa có nơi nào chúng ta sử dụng điều kiện $T(0) = 0$, có nghĩa là kết quả $T(n) \in \Theta(n2^n)$ sẽ hợp lệ với bất kỳ giá trị nào của $T(0)$. Tất nhiên, chúng ta không thể mong đợi điều này xảy ra với mọi trường hợp phụ thuộc loài lặp lại (1.22). Nhớ lại rằng $c_{2,1}$ không thể thu được bằng cách cân bằng các hệ số trong (1.26) và do đó phụ thuộc vào $T(0)$. Và $c_{2,1}$ có thể là hệ số đứng trước hàm phát triển nhanh nhất, trong đó $T(0)$ là vẫn đề quan trọng.

Bài tập

► 1.142. Xác định độ phức tạp của một thuật toán được đưa ra bởi sự phụ thuộc lặp lại:

- a) $T(1) = 1, T(n) = 4T(n - 1) - 2n, n \geq 2$
- b) $T(1) = 0, T(n) = 2T(n - 1) + n + 2n, n \geq 2$
- c) $T(0) = 1, T(n) = 2T(n - 1) + n, n \geq 1$
- d) $T(1) = 1, T(n) = 2T(n - 1) + 3n, n \geq 2$
- e) $T(1) = 2, T(2) = 3, T(n) = 2T(n - 1) - T(n - 2), n \geq 3$

1.15. Các kỹ thuật đặc biệt để phân tích thuật toán

1.15.1. Sử dụng phong vũ biểu

Hãy xem lại đoạn chương trình sau:

```
unsigned sum = 0;
for (i = 0; i < n; i++)
    for (j = 0; j < i*i; j++)
        sum++;
```

Chúng ta đã xác định độ phức tạp của nó ở trên bằng cách sử dụng các thuộc tính của tổng. Chúng ta có thể tiếp cận nó theo cách

khác: chúng ta chọn lệnh thích hợp (phong vũ biểu) và theo dõi số lần nó được thực thi. Điều này giải phóng chúng ta khỏi lo lắng về việc phân tích tất cả các hướng dẫn khác không liên quan đến hướng dẫn đã chọn. Làm thế nào để chọn một phong vũ biểu? Đây phải là một lệnh được thực thi ít nhất thường xuyên như bất kỳ lệnh nào khác trong chương trình. Trong đoạn chương trình trên, một ứng cử viên thích hợp cho điều này là sum ++ . Nhân tiện, giá trị của tổng biến sau khi thực thi đoạn sẽ cho chúng ta số lần thực thi lệnh sum ++

1.15.2. Phân tích khẩu hao

Khi phân tích các thuật toán máy tính, chúng ta thường kiểm tra hành vi của chúng trong trường hợp xấu nhất hoặc trung bình, và chúng ta hầu như không bao giờ quan tâm đến cách chúng hoạt động tốt nhất. Một kỹ thuật phổ biến được sử dụng để phân tích một thuật toán nhằm xác định độ phức tạp của nó trong trường hợp xấu nhất là giả định rằng tại mỗi bước của nó, sự trùng hợp tồi tệ nhất có thể xảy ra. Điều này cho chúng ta độ phức tạp chính xác $O(\dots)$, nhưng không phải lúc nào cũng cung cấp cho chúng ta một ước lượng chính xác cho $\Theta(\dots)$: kết quả thường khá bi quan và trong thực tế, thuật toán hoạt động nhanh hơn nhiều ngay cả trong trường hợp xấu nhất. Lý do là không phải lúc nào trường hợp xấu nhất cũng có thể xảy ra ở mọi bước. Điều này thường đòi hỏi những trường hợp nhẹ hơn vài lần. Do đó, độ phức tạp tổng thể có thể tốt hơn đáng kể so với dự đoán. Hãy xem xét chức năng sau đây làm ví dụ:

Chu kỳ lồng nhau

```
void addOne(char c[], unsigned m)
{
    int i;
    for (i = 0; 1 != c[i] && i < m; i++)
        c[i] = 1 - c[i];
}
```

Hàm nhận dưới dạng tham số một mảng có m phần tử chứa các số không và đơn vị, và coi nó như một bản ghi nhị phân của một số tự nhiên, chữ số nhỏ nhất nằm trong phần tử 0. Nó thêm một đơn

vì, trong mảng là bản ghi nhị phân của số lượng tăng lên. Trong trường hợp tràn, nhận được 0. Độ phức tạp của thuật toán là gì? Rõ ràng, trường hợp khó nhất là khi số chứa m đơn vị (tức là khi xảy ra tràn), vì khi đó tất cả các phần tử của mảng đều được chuyển qua. Độ phức tạp trong trường hợp này rõ ràng là $O(m)$ và $\Theta(m)$, tương ứng. Giả sử chúng ta sử dụng `addOne()` như một bộ đếm và gọi nó n lần, với giá trị $n = 2^m - 1$. Tổng thời gian phức tạp là bao nhiêu? Giả sử ở mỗi bước là trường hợp xấu nhất, chúng ta đạt đến độ phức tạp $O(n.m)$, tức là $O(n.\log_2 n)$. Đây là một đánh giá đúng, nhưng phân tích kỹ hơn cho thấy chúng ta không thể lặp nào tiếp cận được một vụ án nghiêm trọng như vậy. Chúng ta sẽ lưu ý rằng trong quá trình đếm, chúng ta nhận được mọi số nhị phân trong khoảng $[0; 2^m - 1]$ đúng một lần. Một nửa số này là số chẵn và đối với chúng, chúng ta sẽ chỉ có một vòng quay mỗi chu kỳ. Một phần tư số còn lại (lẻ) có bit áp chót được đặt thành 0 và đối với chúng, chúng ta sẽ có hai lần quay mỗi chu kỳ. $1/8$ số khác sẽ có 0 ở bit áp chót (và 1 ở hai bit cuối cùng) và sẽ yêu cầu ba lượt mỗi chu kỳ, v.v. Và chỉ trong một trường hợp, chúng ta sẽ có 1 ở tất cả các vị trí, tức là trường hợp càng nặng thì càng ít xảy ra. Chúng ta để bạn đọc chứng tỏ rằng tổng số vòng quay của chu trình là $2n - 1$. Một suy luận khả thi khác lại dẫn chúng ta đến cùng một kết quả tiệm cận là chúng ta có thể đếm đến n với thời gian $\Theta(n)$, đây là một ước lượng chính xác hơn nhiều.

1.15.3. Định lý cơ bản

Trong phân tích các thuật toán chia để trị (xem Chương 7), các phụ thuộc lặp lại thường phát sinh tùy thuộc vào loại:

$$T(n) = a.T(n/b) + c.n^k$$

Định lý 1.10. Gọi là sự phụ thuộc hồi quy $T(n) = a.T(n/b) + c.n^k$, $n > n_0, a \geq 1, b > 1, k \geq 0, c > 0, n_0 \geq 1$ và a, b, k, c, n_0 là các số nguyên. Giải pháp của nó được đưa ra bởi công thức:

$$T(n) = \begin{cases} \Theta(n^k) & a < b^k \\ \Theta(n^k \log n) & a = b^k \\ \Theta(n^{\log_k a}) & a > b^k. \end{cases}$$

Ví dụ 1.21. Xem xét sự phụ thuộc thường xuyên:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 3T(n/2) + n, n \geq 2. \end{aligned}$$

Sử dụng Định lý 1.10, ta thu được: $a = 3, b = 2, c = 1, k = 1$. Ta có: $3 = a > b^k = 2$. Do đó ta rơi vào trường hợp thứ ba và trực tiếp thu được độ phức tạp $\Theta(n^{\log 3})$.

Ví dụ 1.22. Xem xét sự phụ thuộc thường xuyên:

$$\begin{aligned} T(1) &= d \\ T(n) &= 4T(n/2) + n^2, n \geq 2. \end{aligned}$$

Sử dụng Định lý 1.10, ta được: $a = 4, b = 2, c = 1, k = 2$. Ta có: $4 = a = b^k = 2^2$. Từ đây chúng ta rơi vào trường hợp thứ hai và thu được trực tiếp độ phức tạp $\Theta(n^2 \cdot \log_2 n)$. Lưu ý rằng giá trị của d là bao nhiêu không quan trọng.

Ví dụ 1.23. Xem xét sự phụ thuộc thường xuyên:

$$\begin{aligned} T(1) &= d \\ T(n) &= 2T(\lceil \sqrt{n} \rceil) + \log_2 n, n \geq 2. \end{aligned}$$

Ta đặt $m = \log_2 n$ và nhận được:

$$T(2^m) = 2T(2^{m/2}) + m$$

Bây giờ chúng ta đặt $S(m) = T(2^m)$:

$$S(m) = 2S(m/2) + m$$

Bây giờ chúng ta có thể sử dụng Định lý 1.10: $a = 2, b = 2, c = 1, k = 1$. Ta có: $2 = a = b^k = 2^1$. Ta rơi vào trường hợp thứ hai và nhận được $\Theta(m \cdot \log_2 m)$. Chúng ta quay lại các giả định:

$$T(n) = T(2^m) = S(m) = \Theta(m \cdot \log_2 m) = \Theta(\log_2 n \cdot \log_2(\log_2 n))$$

Bài tập

► **1.143.** Xác định độ phức tạp của một thuật toán nếu được cung cấp bởi sự phụ thuộc đệ quy:

- a) $T(1) = 2, T(n) = 4T(n/3) + n^{\log_3 4}, n \geq 2.$
- b) $T(1) = 0vT(2n+1) = T(2n) = T(n) + \log_2 n, n \geq 2.$
- c) $T(1) = 4vT(n) = 2T(\lceil \sqrt{n} \rceil) + \log_2 n, n \geq 2.$

1.15.4. Các vấn đề về ký hiệu tiệm cận

Mặc dù chúng cho phép chúng ta đánh giá mức độ phức tạp của các thuật toán và chương trình trên cơ sở lý thuyết đúng đắn, chúng ta nên hơi nghi ngờ về các ước lượng tiệm cận. Đặc điểm chính của họ là họ quan tâm đến hành vi của thuật toán với mức tăng n không giới hạn. Tuy nhiên, không có số lượng lớn vô hạn trong thế giới máy tính thực. Điều này có nghĩa là ước lượng do hàm tiệm cận cung cấp cho chúng ta có thể không đủ: ví dụ, vì nó ẩn các hằng số, hoặc vì chúng ta không quan tâm đến các giá trị lớn như vậy của n . Giả sử chúng ta muốn tính giá trị của một hàm, nhưng chỉ đối với các đối số trong phạm vi $[0, 65535]$ và chúng ta có hai thuật toán: tuyến tính ($200000n$) và bậc hai ($2n^2$). Về mặt tiệm cận, thuật toán tuyến tính sẽ nhanh hơn thuật toán bậc hai. Điều này đúng, nhưng từ một nơi nào đó trở đi. Thật không may cho chúng ta đây là $n = 100000 > 65535$. Vì vậy, đối với tất cả các đối số thực tế của hàm, thuật toán bậc hai sẽ nhanh hơn.

Trong tương lai, chúng ta sẽ thường xuyên đối mặt với vấn đề này và học cách sử dụng nó. Ví dụ, sắp xếp nhanh (xem 3.1.6.) Trong trường hợp giữa có độ phức tạp $\Theta(n \log_2 n)$ và sắp xếp chèn (xem 3.1.2.) Có $\Theta(n^2)$. Đối với các giá trị đủ lớn của số phần tử được sắp xếp n , việc sắp xếp nhanh sẽ nhanh hơn nhiều, nhưng đối với 10-20 phần tử, sắp xếp bằng cách chèn là tốt hơn. Thoạt nhìn, điều này hầu như không quan trọng, bởi vì với số lượng phần tử nhỏ như vậy, việc sắp xếp nhanh vẫn nhanh ở mức chấp nhận được và việc sử dụng tính năng phân loại chèn là điều khó chấp nhận. Tuy nhiên, xem xét kỹ hơn sẽ thấy rằng phân loại nhanh liên tục nhín vào các phân vùng nhỏ, với khoảng cách tích tụ. Do đó, hai thuật toán thường được kết hợp để đạt được sự cải thiện đáng kể.

Bài tập

- **1.144.** Ba thuật toán có độ phức tạp đã cho là: $5n^2 - 7n + 13$, $3n^2 + 15n + 100$ và $1000n$. Đổi với mỗi thuật toán để xác định một khoảng giá trị n mà nó nhanh hơn hai giá trị còn lại.
- **1.145.** Lời phê bình về ký hiệu tiệm cận có chính đáng không?

1.16. Các câu hỏi và bài tập

1.16.1. Các bài toán về số, chuỗi, hàm

Sử dụng các bài toán được nêu dưới đây, ngoại trừ để giải quyết vấn đề toán học cụ thể và thử nghiệm với các sơ đồ triển khai khác nhau (lặp lại, đệ quy). Xác định độ phức tạp của thuật toán của bạn và sử dụng nó để dự đoán tốc độ chuyển đổi có thể xảy ra đối với các đầu vào khác nhau.

- **1.146.** Viết chương trình tìm tất cả các số bằng tổng các thừa số của các chữ số của nó.

Gợi ý: Điều quan trọng là phải thu hẹp phạm vi (để tìm giới hạn trên) mà tìm kiếm những con số như vậy. Ví dụ: không có ý nghĩa gì khi kiểm tra các số có 9 chữ số trở lên vì $9, 9! = 3265920$, rõ ràng là nhỏ hơn bất kỳ số nào có 9 chữ số.

Định nghĩa 1.33. Hai số nguyên tố p và q được gọi là "sinh đôi" nếu $p = q + 2$. Người ta chứng minh rằng có vô số cặp số nguyên tố "sinh đôi". 4 đầu tiên trong số đó là $\{3, 5\}$; $\{5, 7\}$; $\{11, 13\}$; $\{17, 19\}$.

- **1.147.** Tìm n cặp số nguyên tố đầu tiên của các cặp sinh đôi.

Định nghĩa 1.34. Tổng của dãy giá trị nghịch đảo của các số sinh đôi

$$S = 1/3 + 1/5 + 1/5 + 1/7 + 1/11 + 1/13 + 1/17 + 1/19 + \dots$$

là một hằng số được gọi là hằng số Bran và xấp xỉ bằng $1,902160578$.

- **1.148.** Tìm tổng các số nghịch đảo của n số nguyên tố đôi một.

► 1.149. Phương trình Diophantine của 2 biến

Phương trình $ax + by = c$ được cho trước, trong đó a, b và c là các số nguyên. Tìm x và y nếu chúng là số nguyên.

Gợi ý: Gọi d là ước chung lớn nhất của a và b . Nếu d không chia c , thì không có lời giải cho bài toán. Sử dụng thuật toán Euclid nâng cao để tìm x và y . Bạn có thể tìm thêm giải pháp nếu bạn có một giải pháp cho vấn đề (x_0, y_0) [Rakhnev, Garov, Gavrilov-1995] không?

► 1.150. $x\%y$

Thực hiện các phép toán của phép chia số nguyên và phần dư của phép chia số nguyên, chỉ sử dụng các phép toán chuẩn $+, -, *, /$ trên các số thực.

► 1.151. Tích của hai số nguyên tố

Một số tự nhiên n đã cho. Tìm số tự nhiên nhỏ hơn n có thể được biểu diễn dưới dạng tích của hai số nguyên tố.

► 1.152. Tổng các số

Với số tự nhiên n tìm số tự nhiên nhỏ nhất $m, m > n$ có tổng các chữ số bằng tổng các chữ số của n . Các chữ số của n được đặt làm phần tử của một mảng và có thể lên đến 2000.

► 1.153. Số lượng không thể

n số nguyên đã cho. Để tìm một trong số chúng mà không thể được biểu diễn bằng tổng của một số những số khác.

► 1.154. $n + 2 = 2m$

Tìm hợp số n nhỏ nhất sao cho $n + 2 = 2m$, với m là số lẻ.

► 1.155. Con số may mắn

Từ danh sách $1, 2, 3, \dots$ mỗi số thứ hai liên tiếp bị loại trừ. Vậy có $1, 3, 5, 7, \dots$. Sau đó, mọi thứ ba bị loại khỏi danh sách mới nhận được ($1, 3, 7, 9, 13$ vẫn còn), v.v. In kết quả sau k số bước.

Gợi ý: Sử dụng một sửa đổi của phương pháp rây.

► 1.156. Cặp đôi nguyên tố cùng nhau

n số tự nhiên đã cho. Tìm số cặp số lớn nhất trong số các số đã cho để các số trong mỗi cặp là nguyên tố cùng nhau.

► 1.157. Đa thức Hermitian

Viết hàm tìm giá trị của đa thức Hermitian $H_n(x)$:

$$H_0(x) = 1;$$

$$H_1(x) = 2x;$$

$$H_n(x) = 2x.H_{n-1}(x) - 2(n-1).H_{n-2}(x), n > 1.$$

► 1.158. Số tribonacci

Số tribonacci được xác định bằng công thức lặp lại sau:

$$F(n) = F(n-1) + F(n-2) + F(n-3), \text{ biết } F(1) = F(2) = F(3) = 1$$

Phần đầu của chuỗi Tribonacci trông như sau:

$$1, 1, 1, 3, 5, 9, 17, \dots$$

Để biên dịch một chương trình, cho trước n , hãy tìm số thứ n của "Tribonacci". Viết thực hiện đệ quy và lặp lại tương ứng. Cái nào trong hai cái hiệu quả hơn và tại sao? Độ phức tạp của thực hiện đệ quy là gì? Hãy tìm công thức để tìm số thứ n của "Tribonacci".

► 1.159. Số Fibonacci của thứ tự p

Số Fibonacci của thứ tự p được định nghĩa là:

$$f_{i+1}^p = f_i^{(p)} + \cdots + f_{i-p}^{(p)}, i \geq p;$$

$$f_p^{(p)} = 1;$$

$$f_i^{(p)} = 0, 0 \leq i \leq p;$$

Như vậy các số Fibonacci thông thường được định nghĩa là các số Fibonacci bậc 1. Viết chương trình tìm k số Fibonacci đầu tiên có bậc p , p và k là các số tự nhiên.

► 1.160. Tổng những nghiệm

Viết chương trình tính tổng $S = \sum_{i=1}^n \frac{1}{\sqrt{i}}$ cho số tự nhiên n cho trước.

Có thể lưu một số phép tính được thực hiện bằng cách triển khai tầm thường không?

► **1.161. Chỗng số đầu tiên**

Số chổng đầu tiên là số tự nhiên có nhiều ước hơn bất kỳ số tự nhiên nào khác trước nó. 6 số đầu tiên của loại này là:

- 1: (có một ước số duy nhất 1)
- 2: (1, 2)
- 4: Có 3 ước (1, 2, 4)
- 6: Có 4 ước số (1, 2, 3, 6)
- 12: Có 6 ước số (1, 2, 3, 4, 6, 12)
- 24: Có 8 ước số (1, 2, 3, 4, 6, 8, 12, 24)

Tìm n số đầu tiên như vậy trên một số tự nhiên n cho trước. Bạn có thể tìm thấy mối quan hệ nào giữa chúng không?

► **1.162. Số nguyên tố dạng $4k + 3$**

Một số tự nhiên n đã cho. Tìm n số nguyên tố đầu tiên có dạng $4k + 3, k \in \mathbb{N}$.

► **1.163. Công thức cho $n!$**

Người ta biết rằng có một công thức để tìm số tiên:

$$S = 1 + 2 + 3 + \cdots + n$$

Tương tự, chúng ta coi tích của n số đầu tiên:

$$P = 1.2.3....n = n!$$

Tìm công thức để tìm $n!$ Điều đó không thực hiện tất cả n phép nhân.

Gợi ý: Nếu bạn không thể tìm ra công thức như vậy trong vài giờ, tốt hơn hết bạn nên từ bỏ: Thực tế, cho đến nay vẫn chưa có ai thành công. Có những công thức, chẳng hạn như công thức Stirling, tìm thấy $n!$ với một ước lượng khá tốt:

$$n! \approx n^n e^{-n} \sqrt{2\pi n}$$

Tuy nhiên, lợi ích của chúng là lý thuyết hơn là thực tế.

► **1.164. Các khoảng với tổng n**

Với một số tự nhiên n đã cho, hãy tìm khoảng $[a, b]$ sao cho tổng các số

$$a + (a + 1) + (a + 2) + \cdots + b$$

bằng n . Ví dụ, với $n = 1986$, một số khoảng thỏa mãn là:

$$[160, 171], [495, 498] \text{ và } [661, 663].$$

► 1.165. *Hàm $\varphi(n)$*

Cho n là một số tự nhiên. Theo $\varphi(n)$, chúng ta có nghĩa là số các số nhỏ hơn n nguyên tố cùng nhau với n . Tìm $\varphi(n)$ với một số nguyên dương n cho trước. Giải thích của bạn cho thực tế là với $n > 2\varphi(n)$ luôn là một số chẵn? Cố gắng tìm công thức chính xác cho $\varphi(n)$.

► 1.166. *Bộ ba Pythagore*

Một số tự nhiên n đã cho. Tìm tất cả các bộ ba Pitago của các số a, b, c (số tự nhiên), $c < n$, trong đó $a^2 + b^2 = c^2$.

► 1.167. *Palindrome bậc*

Cho là một số tự nhiên n . Palindrome bậc $P(n)$ của n được định nghĩa như sau:

1) Nếu n là palindrome thì $P(n) = 1$.

2) Nếu n không phải là palindrome, thì $P(n) = 1 + P(s)$, với s là tổng của n với ảnh phản chiếu của nó, tức là với n , viết qua lại.

Ví dụ: độ palindrome của 36 là 2. Đối với hai bước, chúng ta nhận được palindrome:

$$36 + 63 = 99$$

Tương tự $P(48) = 3$:

$$\begin{aligned} 48 + 84 &= 132 \\ 132 + 231 &= 363 \end{aligned}$$

Tìm $P(n)$ với n bất kỳ. Hãy thử chương trình của bạn với tất cả các số tự nhiên từ 1 đến 250. Nó tìm thấy gì cho trường hợp đặc biệt $P(196)$?

► 1.168. *Bộ tứ Pythagore*

Một số tự nhiên n đã cho. Tìm n (n là số tự nhiên) các nhóm nguyên a, b, c, d khác nhau sao cho:

$$a^2 + b^2 + c^2 = d^2$$

Ví dụ: $(1, 2, 2, 3)$.

► 1.169. Bộ ba số tự nhiên

Một số tự nhiên n đã cho. Tìm n (n là số nguyên) các bộ ba khác nhau của các số tự nhiên a, b, c sao cho $\sqrt{a^2 + b^2}, \sqrt{a^2 + c^2}, \sqrt{b^2 + c^2}$, cũng là các số tự nhiên.

► 1.170. Tổng ngắn nhất của các lập phương

Một số tự nhiên n đã cho. Tìm biểu diễn của n dưới dạng tổng của các hình lập phương bằng cách sử dụng số lượng nhỏ nhất mà các vật sưu tầm được. Ví dụ:

$$567 = 8^3 + 3^3 + 3^3 + 1^3 \text{ (độ dài 4),}$$

nhưng cũng có một giải pháp ngắn hơn:

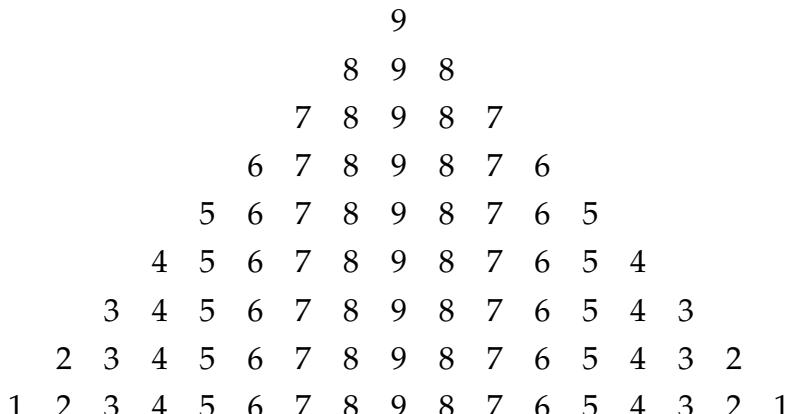
$$567 = 7^3 + 6^3 + 2^3$$

► 1.171. Biểu diễn 2^n

Tìm n nhỏ nhất để chín chữ số đầu tiên của số 2^n là 123454321.

► 1.172. Kim tự tháp số

Viết hàm sao cho một số tự nhiên n cho trước, hiển thị một hình chóp có chiều cao n thuộc loại 1 hoặc 2 (xem Hình 1.16-1.17).



Hình 1.16. Kim tự tháp loại 1

Hình 1.17. Kim tự tháp loại 2

► 1.173. *Quảng trường*

Viết một hàm, cho trước một số tự nhiên n , hiển thị một hình vuông có chiều cao n (xem Hình 1.18).

Hình 1.18. Quảng trường

1.16.2. Bài toán ma trận và bài toán chung

► 1.174. Hình vuông ma thuật

Hình vuông ma thuật của hàng n là một bảng có kích thước $n \times n$,

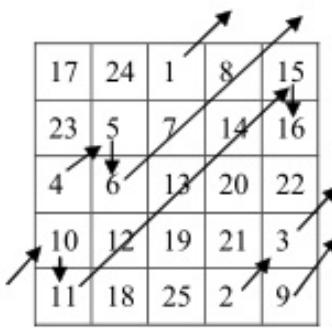
trong mỗi ô có một số tự nhiên từ 1 đến n^2 được viết sao cho tổng các số được viết trên mỗi đường ngang, dọc hoặc đường chéo chính là như nhau và bằng $n \frac{n^2 + 1}{2}$. Ví dụ, với $n = 5$, một hình vuông ma thuật khả dĩ được cho trong Hình 1.19.

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

Hình 1.19. Hình vuông ma thuật

Gợi ý: Đối với n lẻ ($n \geq 3$), một thuật toán khả thi để điền là như sau:

- 1) Chúng ta bắt đầu từ giữa dòng đầu tiên và viết số 1 vào đó.
- 2) Ở mỗi bước, chúng ta di chuyển theo đường chéo lên trên và sang phải và viết số tiếp theo. Nếu chúng ta "đi ra ngoài" bên ngoài bảng trên, chúng ta di chuyển đến hàng cuối cùng, và nếu chúng ta "đi ra ngoài" bên phải, chúng ta di chuyển đến trụ đầu tiên của nó. Nếu trong quá trình di chuyển, chúng ta thấy mình ở trong một ô đã được lấp đầy, thì chúng ta "đi xuống" ô bên dưới nó. Hình 1.20 hiển thị cách một phần của điền bắt đầu, bắt đầu từ 1:



Hình 1.20. Xây dựng một hình vuông kỳ diệu.

Đối với n chẵn, cũng có các thuật toán để thu được một hình vuông ma thuật, nhưng chúng phức tạp hơn một chút [Phép thuật-1].

► 1.175. *Cờ vua*

Xác định số quân cờ tối đa có thể đặt trên một bàn cờ có kích thước $m \times n$ (n, m - các số tự nhiên cho trước) để không có hai quân nào bị tấn công. Để giải quyết công việc tương tự, nếu con số được đặt là: vua; đứng đầu; ngựa; nhân viên văn phòng.

► 1.176. *Từ bốn chữ cái*

Cho trước hai từ gồm bốn chữ cái A_0 và A_n và một từ điển có bốn chữ cái A_i , với $i = 1, 2, \dots, n - 1$. Có thể rút ra từ A_0 từ từ A_n bằng một chuỗi các từ trong từ điển không:

$$A_0 - A_1 - A_2 - \dots - A_n,$$

vì cứ hai từ liên tiếp A_i và A_{i+1} , cho $i = 0, 1, \dots, n - 1$, có đúng một chữ cái khác nhau. Kiểm tra quyết định của bạn với các từ "bay" và "voi" và một từ điển phù hợp.

► 1.177. *Ma trận chuyển đổi*

Một ma trận số nguyên $A_{n \times n}$ với các phần tử a_{ij} được cho trước. Tìm ma trận chuyển vị của nó $A' = \{a'_{ij}\}$ của ma trận đã cho, $a'_{ij} = a_{ji}$, với $1 \leq i, j \leq n$.

1.16.3. Bài toán tổ hợp

Trong một số tác vụ tiếp theo, bạn muốn tạo các đối tượng có các thuộc tính nhất định, chẳng hạn (xem Bài toán 1.43.) Số PIN hợp lệ. Trong các tác vụ như vậy, không nên tạo cấu trúc tổ hợp tổng quát hơn (các biến thể có lặp lại cho ví dụ đã chọn) và sau đó loại trừ các đối tượng không hợp lệ, nhưng nên biên dịch một lược đồ chỉ tạo các đối tượng hợp lệ.

► 1.178. *Số điện thoại*

Năm 1930, một nhóm nghiên cứu từ Phòng thí nghiệm Bell được giao nhiệm vụ đề xuất cách thiết kế số điện thoại của Hoa Kỳ. Chúng được yêu cầu phải có giá trị ít nhất cho đến cuối thế kỷ 20. Sau khi ước tính sẽ có bao nhiêu trại điện thoại, các nhà nghiên cứu tập trung

vào sơ đồ số điện thoại sau:

$(a_1a_2a_3) - b_1b_2b_3 - c_1c_2c_3c_4$, trong đó

$$a_1 \in \{2, 3, \dots, 9\}$$

$$a_2 \in \{0, 1\}$$

$$a_3 \in \{0, 1, \dots, 9\}$$

$$b_1, b_2 \in \{2, 3, \dots, 9\}$$

$$b_3, c_1, c_2, c_3, c_4 \in \{0, 1, \dots, 9\}$$

Tìm số điện thoại khác nhau có thể có được từ sơ đồ trên.

► 1.179. *Hình vuông La tinh*

Hình vuông Latinh của hàng n là một bảng có kích thước n sao cho:

- trong mỗi ô của bảng có một số tự nhiên từ 1 đến n.
- Các số từ mỗi hàng và mỗi cột là hoán vị của các số từ 1 đến n.

Ví dụ, với $n = 4$, hình vuông của Hình 1.21. là tiếng Latinh.

1	2	3	4
2	3	4	1
3	4	1	2
4	1	2	3

Hình 1.21. Hình vuông La tinh

Để tạo một hình vuông Latinh bởi một số tự nhiên n cho trước.

► 1.180. *Tích không đổi*

$2n$ số tự nhiên đã cho. Biết rằng có đúng m cặp số (a_i, b_i) sao cho tích $a_i \cdot b_i$ bằng chính số đó (với $i = 1, 2, \dots, m$). Tìm và in ra tất cả m các cặp số như vậy.

► 1.181. *Ăn tối tại Princess Anna*

Công chúa Anna dự định mời 15 người bạn của mình đi ăn tối. Trong 35 ngày, họ sẽ đến thăm cô ấy đúng 3 lần một ngày. Có thể chọn ba người để trong mỗi 35 ngày, Anna ăn tối với ba người khách khác nhau không? Tạo một "lịch trình" khả thi.

► 1.182. *Giao điểm của các đường chéo trong một n -giác đều.*

Cho số tự nhiên $n, n > 2$. Tìm số giao điểm của các đường chéo

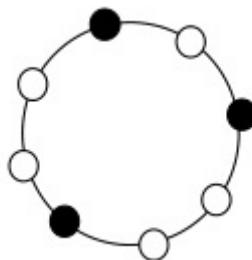
trong một vòng lặp n đều. Ví dụ, không có đường chéo nào trong một tam giác và số này bằng không. Hình vuông có hai đường chéo và tương ứng là một giao điểm, và đối với một hình ngũ giác đều, con số này là 5.

► 1.183. *Bóng trong hộp*

$2t + 1$ hòn bi được cho và phải chia đều vào ba hộp sao cho tổng số bi ở hai hộp bất kì lớn hơn số bi ở hộp thứ ba. Viết một chương trình tạo ra tất cả các phân phối bóng có thể có để đáp ứng điều kiện trên. Tìm công thức cho số lượng các phân phối này.

► 1.184. *Chuỗi hạt*

Một chiếc vòng cổ bao gồm các hạt trắng và đen sao cho không có hai hạt đen liền nhau (xem Hình 1.22.).



Hình 1.22. Vòng cổ

Nếu n là số hạt đen và k là số hạt trắng thì số vòng cổ khác nhau có thể được tạo thành từ các hạt là bao nhiêu? Viết chương trình tạo t dây chuyền khác nhau, cho trước một số tự nhiên t .

► 1.185. *Chiếc cầu*

Viết chương trình chia bài soi cầu: 4 người chơi, mỗi người 13 lá. Trừu tượng thực tế, hãy viết chương trình của bạn theo cách sao cho mỗi lần chơi liên tiếp, một ván bài duy nhất được tạo ra - không có người chơi nào nhận được hai lá bài giống nhau.

► 1.186. *Hình vuông trong lưới*

Một lưới hình vuông với kích thước $n \times n$ được đưa ra. Số ô vuông trong lưới là một hàm của n : tức là tổng số các ô vuông khác nhau có kích thước $1 \times 1, 2 \times 2, \dots, n \times n$?

► 1.187. Các biến trong C

Số tên biến hợp lệ trong C có dưới k ký tự, k là số tự nhiên là bao nhiêu?

► 1.188. Số có n chữ số

Tìm công thức số các số có n chữ số ($n < 11$) chia hết cho k nếu không được phép lặp lại các chữ số. Để đề xuất một thuật toán để tạo thứ tự từ vựng của tất cả các số như vậy.

► 1.189. Khoảng cách giữa các quân cờ

Tìm cách xếp 9 quân trên bàn cờ 8×8 sao cho khoảng cách giữa hai quân là khác nhau, hoặc chứng tỏ rằng điều này là không thể. "Khoảng cách giữa hai hình" đề cập đến số lượng trường trên đường đi ngắn nhất giữa chúng (không có chuyển đổi đường chéo).

► 1.190. *MJ-hợp đồng*

n đôi vợ chồng được đưa ra. Có bao nhiêu cách có thể chia người thành từng nhóm *k* (*k* chẵn, *k* chia hết *n*) sao cho trong mỗi nhóm có đúng *k*/2 nữ và *k*/2 nam và không có ai ở cùng nhóm với vợ/chồng của bạn?

► 1.191. Một hacker trẻ

Viết chương trình tạo số thẻ tín dụng hợp lệ (từ một loại thẻ tín dụng - VISA hoặc MasterCard). Sử dụng mô tả sau để biết số thẻ tín dụng hợp lệ:

Số thẻ tín dụng có 16 chữ số trong nhóm bốn chữ số. Hãy kiểm tra xem thẻ tín dụng sau có hợp lệ không:

4 2 0 4 - 5 8 7 6 - 9 0 1 2 - 5 2 3 4

1) Chúng ta chuyển các chữ số của số và nhận được:

4 3 2 5 2 1 0 9 6 7 8 5 4 0 2 4

2) Nhân đôi các số ở vị trí chẵn và nhận được:

4 6 2 10 2 2 0 18 6 14 8 10 4 0 2 8

3) Nếu lấy số sưu tầm lớn hơn 9, ta chia chúng thành hai chữ số riêng biệt. Chúng ta thu thập các số liệu thu được như sau:

$$\begin{aligned} 4 + 6 + 2 + 1 + 0 + 2 + 2 + 0 + 1 + 8 + 6 \\ + 1 + 4 + 8 + 1 + 0 + 4 + 0 + 2 + 8 = 60. \end{aligned} \quad (1.28)$$

4) Nếu tổng chia cho 10 mà không có số dư, thẻ tín dụng vẫn hợp lệ. Nếu số bắt đầu bằng 4, thì loại của nó là *VISA* và nếu nó có tiền tố 51-55, thì đó là *MasterCard*.

► 1.192. Số PIN

Để viết một chương trình tạo số PIN hợp lệ cho phụ nữ.

Gợi ý: Việc xác nhận mã PIN được thực hiện theo sơ đồ sau: các trọng số sau được so sánh trên từng chữ số của chín đầu tiên: 2, 4, 8, 5, 10, 9, 7, 3, 6. Mỗi chữ số của mã PIN được nhân với trọng lượng tương ứng và tính tổng các sản phẩm thu được. Phần còn lại cho phép chia số nguyên của tổng cho 11 là một số kiểm soát và phải trùng với chữ số thứ mười của mã PIN. Đối với phần dư của 0 hoặc 10, chữ số điều khiển phải là 0.

Ngoài ra, vì 6 chữ số đầu tiên đại diện cho ngày sinh nên chúng phải là ngày hợp lệ trong năm. Ví dụ: mã PIN bắt đầu bằng 810229 không hợp lệ vì năm 1981 không phải là năm cao.

Giới tính được xác định bằng chữ số thứ chín: nếu là số chẵn thì mã PIN dành cho nam giới.

► 1.193. Số Palindromes

Số từ đối xứng (palindromes) có độ dài n ký tự, bao gồm các chữ cái Latinh viết thường là bao nhiêu? Viết chương trình tạo tất cả các palindromes có độ dài n .

CHƯƠNG 2

CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN

2.1. Giới thiệu	154
2.2. Danh sách, ngăn xếp và hàng đợi	157
2.2.1. Danh sách	157
2.2.2. Ngăn xếp	159
2.2.3. Hàng đợi	160
2.2.4. Hàng đợi hai đầu	161
2.3. Thực hiện cụ thể cấu trúc trên	161
2.3.1. Danh sách	161
2.3.2. Ngăn xếp	162
2.3.3. Hàng đợi	164
2.4. Thể hiện cấu trúc động	168
2.4.1. Đưa vào một phần tử	169
2.4.2. Cuộn qua một danh sách	169
2.4.3. Đưa vào sau một phần tử được chỉ định bởi một chỉ dẫn đã cho	171
2.4.4. Đưa vào phía trước một phần tử được chỉ định bởi một thư mục	171
2.4.5. Xóa theo khóa mặc định và con trỏ lên đầu danh sách	172
2.4.6. Xóa một phần tử được chỉ định bởi một thư mục	173
2.5. Cây nhị phân	178
2.5.1. Tìm kiếm bằng chìa khóa	183
2.5.2. Thêm vào một đỉnh mới	184
2.5.3. Xóa một đỉnh bằng từ khóa đã cho	184
2.5.4. Thu thập thông tin	189
2.5.5. Bài tập	191
2.6. Cây cân đối	192
2.6.1. Vòng xoay. Cây đỏ và đen	196
2.6.2. B-Cây	198
2.7. Bảng băm (H-bảng)	201
2.7.1. Hàm băm (H-hàm số)	202

2.7.2. Sự xung đột	203
2.7.3. Hàm băm cổ điển	205
2.7.4. Đối phó với xung đột	214
2.7.5. Triển khai bảng băm	218
2.8. Câu hỏi và bài tập	226

"Một chương trình không có vòng lặp và một biến có cấu trúc không đáng để viết."

Epigram trong lập trình

2.1. Giới thiệu

Điều quan trọng nhất đối với giải pháp hiệu quả của mỗi bài toán là sự lựa chọn của một thuật toán thích hợp. Tùy thuộc vào sự lựa chọn này liệu vấn đề sẽ được giải quyết chính xác hay xấp xỉ, một phần hay toàn bộ, thời gian tính toán và nguồn lực cần thiết để giải quyết nó, v.v. Đổi lại, hiệu quả của từng thuật toán cụ thể cũng phụ thuộc vào cách nó sẽ được thực hiện. Sự phức tạp của nó liên quan chặt chẽ đến cách thức mà các đối tượng trong bài toán sẽ được trình bày và tổ chức, nghĩa là với sự lựa chọn *cấu trúc dữ liệu* thích hợp. Để hiểu tại sao lại như vậy, chỉ cần suy nghĩ sâu hơn về công việc của một chương trình máy tính là đủ - ngoài thuật toán đã chọn, phần lớn thời gian tính toán được dành cho các quy trình liên quan đến truy cập, lưu trữ và xử lý dữ liệu.

Một trong những ví dụ đơn giản nhất về cấu trúc dữ liệu là *các biến* kiểu đơn giản (số nguyên/số thực, ký hiệu). Chúng được hỗ trợ bởi hầu hết các ngôn ngữ lập trình và được sử dụng trong hầu hết các chương trình. Cấu trúc của chúng là không đổi - chúng chỉ có thể chấp nhận các giá trị được chấp nhận cho kiểu của chúng và chiếm một lượng bộ nhớ không đổi (nghĩa là không thể thay đổi trong quá trình hoạt động của chương trình).

Một cấu trúc dữ liệu cơ bản khác là *mảng*: một số cố định các biến cùng kiểu được lưu trữ tuần tự trong bộ nhớ máy tính. Việc truy cập vào từng biến này (tức là vào các phần tử của mảng) được thực hiện bởi sự kết hợp giữa tên của mảng và chỉ số của biến tương

ứng. Ưu điểm chính của việc làm việc với mảng là quyền truy cập vào phần tử thứ k là trực tiếp (tuy nhiên, không giống như các biến đơn giản, có địa chỉ gián tiếp: địa chỉ của đầu cộng với phần bù được tính toán). Mặt khác, kích thước của mảng là cố định - điều này có nghĩa là thứ nhất, bộ nhớ được cấp phát khi khai báo mảng vẫn bị chiếm trong suốt chương trình và thứ hai, nếu kích thước của mảng không đủ cho nhu cầu của chương trình, phóng đại chỉ có thể được thực hiện bằng cách sao chép các phần tử vào một mảng lớn.

Các biến và mảng đơn giản được định nghĩa là *cấu trúc dữ liệu tĩnh*. Hầu hết các ngôn ngữ lập trình cũng cung cấp khả năng cấp phát bộ nhớ động. Với nó, bộ nhớ được dự trữ khi cần thiết và được giải phóng khi không còn cần thiết. Việc sao lưu và giải phóng diễn ra trong quá trình hoạt động của chương trình, dẫn đến việc sử dụng bộ nhớ linh hoạt và hiệu quả, cũng như khả năng mô hình hóa các cấu trúc dữ liệu phức tạp hơn có logic riêng và "tuổi thọ" thuật toán của riêng chúng.

Nói chung, việc biểu diễn và mô hình hóa các đối tượng thực trong chương trình máy tính, cũng như các hoạt động được thực hiện trên chúng, có thể được chia thành hai bài toán con:

1) *Định nghĩa cấu trúc dữ liệu trừu tượng* (ADS): đây là cách mà các đối tượng thực sẽ được mô hình hóa thành các đối tượng toán học, cũng như xác định tập hợp các phép toán cho phép trên chúng.

2) *Hiện thực hóa các cấu trúc dữ liệu trừu tượng*: Đây là cách mà các đối tượng toán học đã xác định sẽ được biểu diễn trong bộ nhớ máy tính (thông qua các kiểu đơn giản hoặc dưới dạng kết hợp các triển khai có sẵn của ASD khác), cũng như cách thức mà chúng sẽ được hiện thực hóa hoạt động với chúng.

Nói cách khác, mục đích của việc xác định cấu trúc dữ liệu trừu tượng là xác định những gì có thể được thực hiện với các đối tượng, và việc thực hiện là cách thực hiện.

Chúng ta sẽ minh họa những gì đã được nói cho đến nay bằng một ví dụ cụ thể. Hãy trở thành một bộ bài 52 lá. Để biểu diễn nó, chúng ta có thể sử dụng một cấu trúc dữ liệu trừu tượng - *một tập hợp* có các phần tử là quân bài. Chúng ta xem xét các hoạt động sau:

- *loại trừ* bất kỳ phần tử nào khỏi bộ (loại bỏ bất kỳ thẻ nào khỏi

bộ bài).

- *Đưa vào* một phần tử trong tập hợp (thêm một quân bài).
- *kiểm tra* xem một phần tử có thuộc bộ hay không (kiểm tra xem một thẻ có trong bộ bài hay không).

Việc thực hiện bộ này cũng rất mơ hồ. Ví dụ: mỗi phần tử của nó (bộ bài) có thể được biểu diễn bằng hai ký hiệu - cho loại quân bài ('2', ..., '9', 'T', 'J', 'Q', 'K', ' A ') và cho quân bài vẽ (' S ', ' C ', ' D ', ' H '). Có thể biểu diễn khác - trên mỗi thẻ để so sánh một số tự nhiên duy nhất giữa 0 và 51 (số thẻ khác nhau là 52). Ngoài ra, chúng ta phải chọn một đại diện của tập hợp và thực hiện các phép toán được phép trên nó. Chúng ta sẽ hiển thị một khả năng hiện thực hóa (chúng ta giả định rằng các thẻ được đánh dấu bằng các số nguyên từ 0 đến 51):

Chúng ta sẽ sử dụng một mảng `int cards[52]`. Giá trị của thẻ cards[i] sẽ là 1 nếu thẻ có số i nằm trong bộ bài. Nếu không giá trị của các thẻ cards[i] sẽ là 0. Như vậy, để thêm thẻ có số k, chỉ cần thực hiện các thẻ gán `[k] = 1` là đủ; . Phép toán loại trừ một phần tử tùy ý được thực hiện như sau: chúng ta tìm một số k cho thẻ nào `cards[k] == 1` và gán thẻ `cards[k] = 0`. Việc kiểm tra xem một quân bài có số k có trong bộ bài hay không đưa vào một phép thử duy nhất về giá trị của `cards[k]`.

Vấn đề thực hiện là tối quan trọng trong việc giải quyết bài toán cụ thể. Trong ví dụ về bộ bài, việc thực hiện một tập hợp bằng cách sử dụng một mảng cards[52] là thích hợp, vì các quân bài chỉ có 52. Tuy nhiên, có thể các phần tử của tập hợp đó lớn hơn nhiều. Ví dụ, hãy đánh số các sinh viên trong một trường đại học - mỗi sinh viên được gán một số tự nhiên duy nhất. Nếu chúng ta sử dụng một chuyển đổi tương tự như trong ví dụ bản đồ để biểu diễn số lượng sinh viên trong một nhóm hành chính (giả sử rằng không có hơn 30.000 sinh viên tại trường đại học), sẽ có một sự lãng phí bộ nhớ lớn: toàn bộ một mảng `int students[30000]`, trong khi số học sinh trong một nhóm thực sự không quá 30. Chúng ta có thể tiếp cận theo cách khác: nhập một biến số nguyên `unsigned n`, biểu thị số học sinh trong nhóm và một mảng `unsigned students[n]` sao cho vị trí thứ i ($0 \leq i < n$) để ghi số sinh viên.

Nói chung, việc lựa chọn cách triển khai không chỉ phụ thuộc

vào dung lượng bộ nhớ sẽ được sử dụng, mà còn phụ thuộc vào độ phức tạp tính toán của mỗi hoạt động. Trong chương này, chúng ta sẽ làm quen với các cấu trúc dữ liệu trừu tượng chính, cũng như hai cách tiếp cận quan trọng để triển khai chúng - tuân tự (tính, như chúng ta đã áp dụng trong các ví dụ được liệt kê ở trên) và kết nối (động).

2.2. Danh sách, ngăn xếp và hàng đợi

2.2.1. Danh sách

Định nghĩa 2.1. Một *danh sách tuyến tính* (hay chỉ một danh sách) được gọi là một dãy gồm $n(n \geq 0)$ phần tử x_1, x_2, \dots, x_n , được sắp xếp tuân tự (tuyến tính). Tại $n = 0$, danh sách được gọi là *rỗng*. Nếu $n > 0$, thì x_1 được gọi là *phần tử đầu tiên* trong danh sách và x_n - *phần tử cuối cùng*.

Mối quan hệ duy nhất giữa các phần tử trong cấu trúc này được xác định bởi các điều kiện: Với mỗi i ($1 \leq i \leq n$) phần tử thứ i , x_i đứng trước phần tử x_{i-1} và sau là phần tử x_{i+1} .

Trên thực tế, từ "tuyến tính" là thừa: danh sách phi tuyến tính có tên đặc biệt - cột, cây, v.v., và không được coi là danh sách. Chúng ta sử dụng nó để tuân thủ thuật ngữ được chấp nhận chung, nhưng rất tiếc là sai.

Trong thực tế, bởi "phần tử" trong một danh sách, chúng ta có nghĩa là một cấu trúc dữ liệu (trường) tùy ý. Thông thường, các phần tử của danh sách là các kiểu dữ liệu đơn giản và có cấu trúc giống hệt nhau.

Hãy biểu thị kiểu dữ liệu mà mỗi phần tử chứa dữ liệu và xem xét ví dụ sau: Chúng ta muốn xây dựng một danh sách sinh viên trong một trường đại học. Mỗi mục trong danh sách có thể chứa các thông tin sau:

Cấu trúc dữ liệu

```
struct data {
    short age; /* Tuổi của sinh viên (nhỏ hơn 128) */
    char sex; /* Giới tính sinh viên, 'm' - nam; 'f' - nữ */
    unsigned fn; /* Số báo danh sinh viên */
```

```
char *name; /*Tên sinh viên*/  
};
```

Hãy xem xét các hoạt động chính có thể được thực hiện trên một danh sách tuyến tính:

- Có được quyền truy cập vào phần tử thứ k trong danh sách (và có thể *thay đổi* giá trị của bất kỳ trường nào của nó).
- *Đưa vào* (chèn) một mục trong danh sách (trước hoặc sau một mục, cũng như trong danh sách trống)
- *Loại trừ* (xóa) mục thứ k khỏi danh sách.
- *Tìm* tất cả các mục trong danh sách có chứa một giá trị (trong một trường hoặc tập hợp các trường nhất định).

Nhiều hoạt động khác có thể được định nghĩa - sắp xếp, hợp nhất, lật danh sách, v.v. Các hoạt động này mang tính thuật toán cao hơn và sẽ được thảo luận ở phần sau của cuốn sách, và trong chương này, chúng ta sẽ giới hạn bản thân chỉ xem xét các nguyên tắc cơ bản trong việc xác định và triển khai cấu trúc dữ liệu.

Nói chung, thao tác tìm kiếm có thể được thực hiện theo một tiêu chí khác: ví dụ, trong danh sách sinh viên được xác định ở trên, chúng ta có thể quan tâm đến tất cả sinh viên dưới 22 tuổi hoặc tất cả sinh viên có số giảng viên là số nguyên tố.

Từ bây giờ, khi chúng ta nói về các phần tử của một cấu trúc, để thuận tiện, chúng ta sẽ giả sử rằng cùng với dữ liệu mà nó chứa, mỗi phần tử nhất thiết sẽ có một trường, trường này được gọi là khóa của phần tử. Việc tìm kiếm một mục trong danh sách sẽ chỉ được thực hiện bằng khóa này.

Các hoạt động được xác định ở trên không thể được thực hiện theo cách mà chúng đều có hiệu quả cùng một lúc (ví dụ: tất cả chúng đều có độ phức tạp không đổi). Ngoài ra, các trường hợp $k = 1$ và $k = n$ đặc biệt hơn - có thể truy cập chúng với độ phức tạp ít hơn so với truy cập vào các mục khác trong danh sách. Hơn nữa, thường chỉ cần thực hiện các thao tác trong danh sách trên phần tử đầu tiên và/hoặc phần tử cuối cùng, điều này dẫn đến việc định nghĩa các cấu trúc dữ liệu cụ thể:

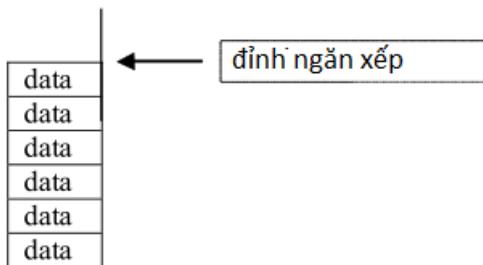
Bài tập:

▷ 2.1. Tại sao không thể thực hiện tất cả bốn hoạt động cơ bản một

cách hiệu quả như nhau, ví dụ, có độ phức tạp liên tục?

2.2.2. Ngăn xếp

Ngăn xếp là một danh sách tuyến tính trong đó tất cả các hoạt động (bật, tắt, v.v.) chỉ được thực hiện ở một đầu, tức là chỉ trên phần tử đầu tiên hoặc chỉ trên phần tử cuối cùng. Người ta chấp nhận rằng phần tử mà các hoạt động được thực hiện được gọi là phần trên cùng của ngăn xếp (xem Hình 2.1).



Hình 2.1. Ngăn xếp

Các phép toán trừu tượng cổ điển trên ngăn xếp S như sau:

Thao tác trên ngăn xếp

```
void init(S); // khởi tạo ngăn xếp trống
void push(S, data x); // thêm một phần tử mới vào đầu ngăn xếp
data pop(S); // lấy vật phẩm từ trên cùng của ngăn xếp
int isEmpty(S); // kiểm tra xem ngăn xếp có trống không
```

Các hoạt động bổ sung của ngăn xếp thường được thực hiện: lấy một phần tử từ trên cùng của ngăn xếp mà không cần tắt nó; thêm/bớt k phần tử, v.v.

Nhiều ví dụ về ngăn xếp có thể được đưa ra. Từ stack dịch sang tiếng Việt là đồng. Sự tương đồng là hiển nhiên - nếu chúng ta nhìn vào một đồng báo (hoặc sách) trong một chiếc hộp, chúng ta có thể dễ dàng lấy tờ báo từ trên cùng hoặc thêm một tờ mới ở trên cùng. Người ta cũng nói rằng ngăn xếp tuân theo kỷ luật LIFO (Last-In-First-Out Vào sau, ra trước).

2.2.3. Hàng đợi

Hàng đợi là một danh sách tuyến tính trong đó thao tác bật chỉ có thể được thực hiện ở một đầu của danh sách và hoạt động tắt - ở đầu kia (Hình 2.2).

Các thao tác cổ điển trên hàng đợi Q như sau:

Thao tác trên hàng đợi

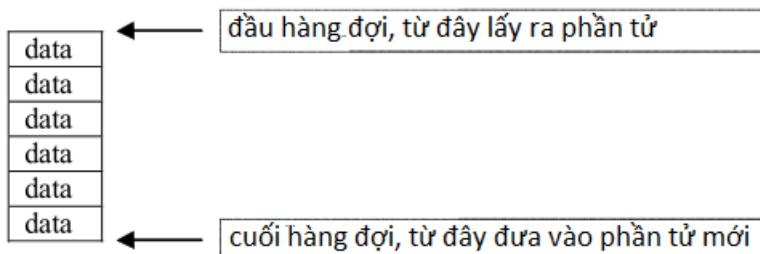
```

void init(Q); //khởi tạo hàng đợi trống
void put(Q, data x); // thêm một phần tử (ở cuối hàng đợi)
data get(Q); //truy xuất phần tử đầu tiên của hàng đợi
                // (trả về phần tử đầu tiên, sau đó xóa nó)
int isEmpty(Q); //kiểm tra xem hàng đợi có trống không

```

Do đó, các phần tử của hàng đợi tuân theo kỷ luật FIFO (First-In-First-Out: vào trước, ra trước) - ngược lại ngăn xếp. Và ở đây chúng ta có một phép tương tự với ý nghĩa của từ xếp hàng được sử dụng trong thực tế - ví dụ: trong hàng đợi mua vé (trừ khi chúng ta đủ đậm), chúng ta xếp hàng ở phía sau và chờ đến lượt của mình, và khi chúng ta ra phía trước, chúng ta được phục vụ và rời khỏi hàng đợi.

Có một số loại hàng đợi thú vị và quan trọng - ví dụ: hàng đợi ưu tiên (xem Bài tập 2.6.)



Hình 2.2. Hàng đợi

Ngăn xếp và đuôi có một số ứng dụng. Trong hầu hết các ngôn ngữ lập trình, phần bộ nhớ của lập trình viên được sử dụng ngầm cho ngăn xếp chương trình, việc tính toán các biểu thức được thực hiện dễ dàng thông qua ngăn xếp, v.v. Các lệnh chờ được thực thi

bởi bộ xử lý tạo thành một hàng đợi; có một hàng đợi cho các tài liệu được gửi đến máy in để in; các hệ thống xử lý đơn mô phỏng việc thực thi song song các tác vụ, sử dụng hàng đợi, v.v.

2.2.4. Hàng đợi hai đầu

Có một phiên bản khác, ít được sử dụng hơn, của danh sách, là bản tóm tắt của một ngăn xếp và một hàng đợi cùng một lúc - đây là một *hàng đợi hai đầu* (DEQue, viết tắt của Double-Ended-Queue). Như tên gọi của nó, trong cấu trúc này, các phần tử có thể được bật và tắt ở cả hai đầu. Về phần mình, hàng đợi hai đầu cũng có một số biến thể.

Có thể chỉ bật ở một bên và tắt - trên cả hai (hang đợi hai đầu có lối vào hạn chế) hoặc ngược lại: bật ở cả hai bên và tắt - ở một (hang đợi hai đầu có lối ra hạn chế). Nhìn chung, các bộ bài không được sử dụng rộng rãi, mặc dù trong một số trường hợp đặc biệt (ví dụ, trong các hệ thống đa xử lý) sẽ rất tiện lợi khi sử dụng cấu trúc cho các bài toán khác nhau (tính toán giá trị của biểu thức, v.v.) [Shishkov-1995].

2.3. Thực hiện cụ thể cấu trúc trên

2.3.1. Danh sách

Cách đơn giản nhất để thực hiện một danh sách là lưu trữ tuần tự các phần tử của nó trong bộ nhớ của máy tính (Hình 2.3). Trong trường hợp này, địa chỉ của phần tử thứ k , $addr(k)$ được xác định bằng công thức

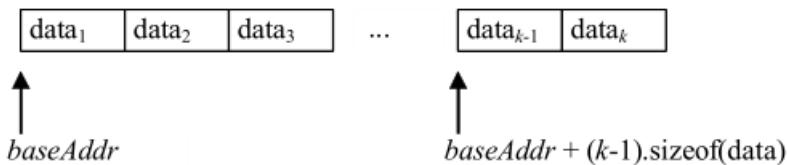
$$addr(k) = addr(k - 1) + sizeof(dliu),$$

trong đó $sizeof(data)$ là bộ nhớ cần thiết để lưu một mục trong danh sách. Trong kiểu triển khai này, quyền truy cập vào phần tử thứ k là trực tiếp. Vì địa chỉ của phần tử ban đầu đã được biết (đây là địa chỉ cơ sở $baseAddr$), nên địa chỉ của phần tử thứ k có thể được xác định bằng công thức, bắt kể địa chỉ của phần tử đứng trước:

$$addr(k) = baseAddr + (k - 1)sizeof(data).$$

Tuy nhiên, sự thuận tiện của việc truy cập trực tiếp vào các phần tử được trả bằng cách bật và tắt không hiệu quả: để đưa vào một phần

tử sau chữ thứ k , chúng ta phải di chuyển tất cả các phần tử từ $k + 1$ đến n theo một vị trí sang phải. Tương tự, để loại trừ phần tử thứ k , chúng ta phải chuyển tất cả các phần tử từ $k + 1$ đến n theo một vị trí sang trái. Tìm kiếm khóa cũng chậm: trung bình $\frac{n}{2}$ so sánh được thực hiện (diều này cũng đúng với việc chèn và xóa), tức là độ phức tạp trung bình là $\Theta(n)$.



Hình 2.3. Danh sách chia bộ nhớ

2.3.2. Ngăn xếp

Chúng ta sẽ triển khai các cấu trúc ngăn xếp và hàng đợi một cách tinh thông qua một mảng. Trong trường hợp ngăn xếp, chúng ta sẽ nhập một chỉ mục duy nhất sẽ cho biết địa chỉ ở trên cùng của ngăn xếp. Do đó, trong hoạt động đưa vào, phần tử sẽ được ghi vào địa chỉ được chỉ định bởi chỉ mục, sau đó chỉ mục sẽ được tăng lên một. Theo đó, khi tắt, chỉ số đầu tiên sẽ giảm đi một, sau đó phần tử sẽ được trích xuất. Đây là cách triển khai C trông như thế nào khi sử dụng mảng stack[] của loại data và một biến top trỏ đến đầu ngăn xếp:

Khung ngăn xếp

```
#include <stdio.h>
typedef int data;
data stack[10];
int top;

void init(void) { top = 0; }
void push(data i) { stack[top++] = i; }
data pop(void) { return stack[--top]; }
int empty(void) { return (top == 0); }
```

```

int main(void) {
    /* ... */
    return 0;
}

```

Những bất lợi của việc thực hiện ở trên là một số. Đầu tiên chúng ta khai báo một mảng có 10 phần tử. Nếu đã có 10 phần tử trong ngăn xếp (index top tương ứng bằng 10) và chúng ta muốn đưa vào một phần tử mới, sẽ xảy ra tràn và chương trình sẽ không đăng ký phần tử đó. Việc ghi lại trên bộ nhớ "ngoại lai" này có thể dẫn đến kết quả không mong đợi (và thường là thảm khốc). Tương tự, khi cố gắng loại trừ một mục khỏi một ngăn xếp trống, chỉ mục top sẽ trở thành một số âm và hàm pop() sẽ trả về một giá trị tùy ý (thực sự trả về giá trị của địa chỉ trong bộ nhớ ngay trước phần tử đầu tiên của mảng stack). Ngoài ra, một sự bất tiện liên tục được tạo ra bởi hằng số 10 được sử dụng trong chương trình. Ví dụ: nếu chúng ta quyết định thay đổi số phần tử tối đa thành 20, chúng ta sẽ phải xem lại toàn bộ mã chương trình, điều này một mặt đòi hỏi thời gian và mặt khác - là một hoạt động không an toàn, bởi vì chúng ta có thể dễ dàng bỏ lỡ một cái gì đó (ví dụ: , đến từ 10–1, v.v.).

Để giải quyết những thiếu sót này, chúng ta sẽ thực hiện các sửa đổi sau - chúng ta sẽ xác định macro MAX, đặt số lượng phần tử tối đa trong ngăn xếp. Khi một mục mới được đưa vào, nó sẽ được kiểm tra xem có phải là top nó đã không trở nên lớn hơn MAX và nếu vậy, thì có một phần tràn. Một thay đổi khác đã được thực hiện trong chương trình bên dưới - trước mỗi lần cố gắng tắt một mục, hãy kiểm tra xem ngăn xếp có trống không.

Ví dụ

Chương trình 2.1. Ngăn xếp (202stack2.c)

```

1 #include <stdio.h>
2 #define MAX 10
3 typedef int data;
4 data stack[MAX];
5 int top;

7 void init(void) { top = 0; }

```

```

9 void push(data i)
10 { if (MAX == top)
11     fprintf(stderr, "Them vao ngan xep \n");
12 else
13     stack[top++] = i;
14 }

16 data pop(void)
17 { if (0 == top) {
18     fprintf(stderr, "Ngan Xep troong \n");
19     return 0;
20 }
21 else
22     return stack[--top];
23 }

25 int empty(void) { return (0 == top); }

27 int main(void) {
28     data p;
29     init();

31 //Đưa vào một số nguyên và đưa vào ngăn xếp, số 0 kết thúc
32     scanf("%d", &p);
33     while (0 != p) {
34         push(p);
35         scanf("%d", &p);
36     };

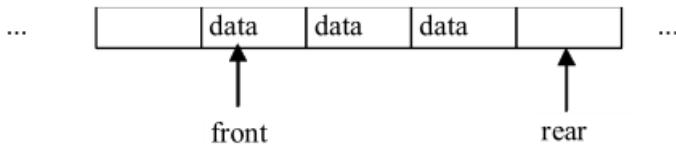
38 //Lấy ra lần lượt các phần tử của ngăn xếp và in ra.
39 //như vậy lấy ra ngược với dãy đưa vào.
40     while (!empty()) printf("%d ", pop());
41     printf("\n");
42     return 0;
43 }

```

2.3.3. Hàng đợi

Việc thực hiện hàng đợi phức tạp hơn một chút. Để đặt trước bộ nhớ tuần tự, chúng ta sẽ lại sử dụng một mảng, nhưng các chỉ mục

sẽ là hai: front , cho biết phần đầu của hàng đợi và rear , cho biết vị trí sau khi kết thúc (Hình 2.4).

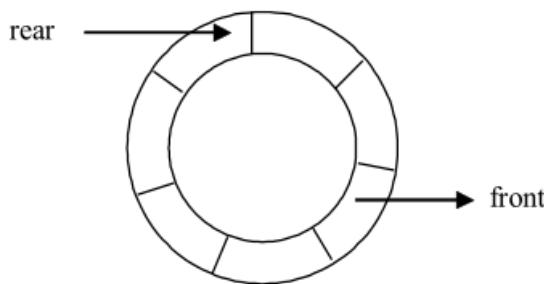


Hình 2.4. Danh sách chia bộ nhớ

Việc bật và tắt các phần tử được thực hiện như sau:

- *Đưa vào phần tử i*: Viết i vào vị trí được chỉ ra bởi chỉ số $rear$ và tăng nó lên một.
- *Loại trừ một mục*: Trả mục đó về phía trước $front$, sau đó tăng chỉ mục $front$ với một.

Có một số vấn đề cần giải quyết khi triển khai bật và tắt. Vấn đề chính liên quan đến thực tế là hàng đợi "di chuyển" trong bộ nhớ: Nếu chúng ta bật và tắt các mục liên tục, thì chỉ mục front và rear chúng sẽ tăng liên tục và nhanh chóng vượt quá bộ nhớ được cấp phát ban đầu cho hàng đợi. Vì vậy, nếu chúng ta sử dụng mảng Queue [MAX] (với một số phần tử cố định MAX), kích thước của nó sẽ nhanh chóng không đủ, mặc dù trong thực tế, hàng đợi sẽ chứa ít hơn MAX thành phần. Do đó, nếu một trong hai chỉ mục front hoặc rear trở thành bằng với MAX , nó sẽ được gán giá trị bằng 0. Do đó chúng ta đạt được "tính chu kỳ" của mảng queue[] (xem Hình 2.5).



Hình 2.5. Hàng đợi thể hiện qua mảng chu kỳ

Tại thời điểm bắt đầu, các chỉ số phía trước và phía sau trỏ đến phần tử không. Hơn nữa, nếu một lúc nào đó chúng trở lại bằng nhau (trỏ đến cùng một ô), điều này có nghĩa là một trong hai điều:

- Nếu bình đẳng thu được sau *không bao gồm* một phần tử (chỉ mục front Has to rear), Thì hàng đợi vẫn trống sau thao tác.
- Nếu đẳng thức nhận được sau *inclusive* của một phần tử (rear Đã đạt tới front), Thì hàng đợi đã đầy - nó chứa phần tử MAX và không thể thêm được nữa.

Đối với trạng thái của hàng đợi (cho dù nó đầy hay trống), chúng ta sẽ sử dụng một biến bổ sung rỗng, bằng 1 khi hàng đợi trống và bằng 0 - nếu không. Biến rỗng được khởi tạo bằng 1 (hàng đợi trống ở đầu) và nó nhận giá trị 0 ngay sau lần đưa vào đầu tiên một phần tử. Hơn nữa, biến này chỉ được sửa đổi trong hai trường hợp đặc biệt được liệt kê ở trên. Sự nhận biết đầy đủ của Si như sau:

Chương trình 2.2. Hàng đợi (203queue.c)

```

1 #include <stdio.h>
2 #define MAX 10

4 typedef int data;
5 data queue[MAX];
6 int front, rear, empty;

8 void init(void) { front = rear = 0; empty = 1; }

10 void put(data i)
11 { if (front == rear && !empty) {
12 //Kiểm tra đầy chưa*/
13 //Đầy rồi thì các chỉ số bằng nhau, hàng đợi khác rỗng
14 fprintf(stderr, "Hang doi rong! \n");
15     return;
16 }
17 queue[rear++] = i;
18 if (rear >= MAX) rear = 0;
19 empty = 0;
20 }

22 data get(void)
23 { data x;
```

```

24  if (empty) { // Kiểm tra hàng đợi rỗng
25      fprintf(stderr, "Hang doi rong! \n");
26      return 0;
27  }
28  x = queue[front++];
29  if (front >= MAX) front = 0;
30  if (front == rear) empty = 1;
31  return x;
32 }

34 int main(void){
35     data p;
36     int i;
37     init();
38     for (i = 0; i < 2 * MAX; i++) {
39         put(i);
40         p = get();
41         printf("%d ", p);
42     }

44     printf("\n");

46 //Đưa phần tử tiếp theo vào
47     for (i = 0; i < MAX + 1; i++) put(i);

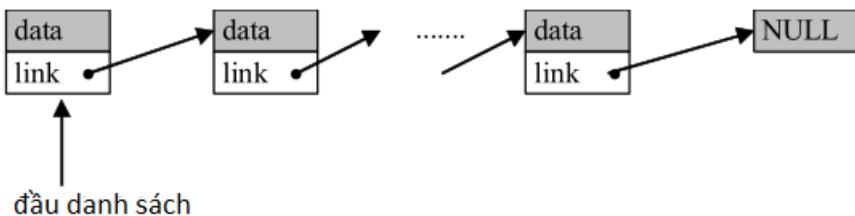
49 //Lấy các phần tử ra, khi có lỗi là hàng đợi rỗng
50     for (i = 0; i < MAX + 1; i++) get();
51     return 0;
52 }
```

Bài tập

- ▷ 2.2. Thiết kế lại các chương trình trên để các hàm không in thông báo lỗi, nhưng trả về giá trị Boolean nếu chúng thành công.
- ▷ 2.3. Để thực hiện phiên bản "bán tĩnh" của các chương trình trên: nếu ngăn xếp / hàng đợi đã đầy, để tự động mở rộng mảng.
- ▷ 2.4. Để thực hiện các chức năng chính để làm việc với Dec.

2.4. Thể hiện cấu trúc động

Với cấu trúc dữ liệu động, các phần tử không nhất thiết phải nằm trong các địa chỉ bộ nhớ liên tiếp. Trong trường hợp chung, chúng ta chỉ có một con trỏ đến đầu (phần tử đầu tiên) của danh sách và địa chỉ của tất cả các phần tử khác không thể truy cập trực tiếp. Sau đó chúng được truy cập như thế nào? Một trường khác được thêm vào mỗi phần tử (ngoài các trường chứa dữ liệu): một con trỏ đến phần tử tiếp theo (Hình 2.6).



Hình 2.6. Danh sách liên kết động

Loại danh sách tuyến tính này được gọi là danh sách liên kết đơn tuyến tính: liên kết đơn bởi vì chúng ta có một con trỏ chỉ đến phần tử tiếp theo.

Để đến phần tử thứ k của danh sách, chúng ta phải đi qua tất cả các phần tử đứng trước thứ $k - 1$ một cách tuần tự. Cấu trúc của các phần tử như sau:

Cấu trúc động

```

typedef int data;
typedef int keyType;
struct list {
    keyType key;
    data info;
    struct list *next;
};

```

Mỗi phần tử có một định danh: key (field keyType key). info chứa dữ liệu bổ sung cho mỗi phần tử. Con trỏ đến phần tử tiếp theo là một con trỏ đến cùng cấu trúc **struct** list *next; - Ngôn ngữ này tự

cho phép định nghĩa đệ quy cấu trúc như vậy (nhưng chỉ khi chúng ta sử dụng con trỏ tới một cấu trúc; khai báo tĩnh kiểu **struct** list next; Trong phần thân của list chưa đủ). Mục cuối cùng trong danh sách liên kết động có giá trị là NULL : gán một giá trị như vậy cho thư mục *next có nghĩa là không còn mục nào trong danh sách.

Có các cách triển khai khác có thể có của danh sách tuyến tính [Nakov-1998] [Wirth-1980]. Ví dụ: trong danh sách liên kết đôi, hai con trỏ được giữ cho mỗi phần tử: phần tử trước và phần tử tiếp theo. Nếu trong danh sách liên kết đơn có thêm một con trỏ từ phần tử cuối cùng đến phần tử đầu tiên, thì danh sách đó được gọi là tuần hoàn và các phần tử khác.

Để sử dụng danh sách liên kết, chúng ta khai báo:

struct list *L;

và khởi tạo L = NULL. Thao tác đầu tiên chúng ta sẽ xem xét là đưa một mục vào đầu (trước mục đầu tiên) của danh sách.

2.4.1. Đưa vào một phần tử

Ba hoạt động đơn giản được thực hiện liên tiếp (xem Hình 2.7):

1) Cấp phát bộ nhớ cho một phần tử mới **struct** list:

struct list *temp;

temp = (**struct** list *) malloc (**sizeof** (*temp));

2) Phần tử tạm thời mới cho biết phần đầu của danh sách:

temp->tiếp theo = L;

3) Phần đầu của danh sách được thiết lập trong phần tử mới và trường dữ liệu và khóa đã đặt được gán cho nó:

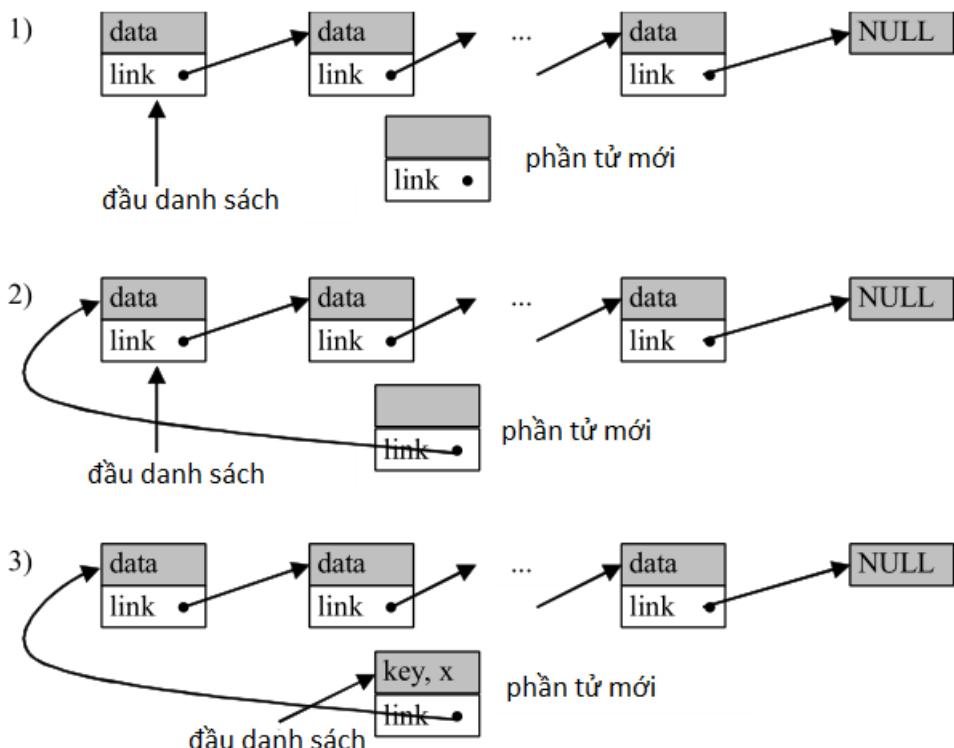
L = temp;

L->key = key;

L->info = x;

2.4.2. Cuộn qua một danh sách

Với thao tác này, có thể xây dựng một danh sách toàn bộ. Nó có thể được thu thập thông tin như sau: chúng ta bắt đầu từ chỉ mục đến đầu danh sách L và chúng ta sẽ in khóa của các phần tử khi thu thập thông tin:



Hình 2.7. Đưa vào đầu và dịch chuyển con trỏ

```
while (L != NULL) {
    printf("%d ", L->key);
    L = L->next; //chuyển sang mục tiếp theo
}
```

Thu thập thông tin tương tự được sử dụng khi tìm kiếm một phần tử bằng một khóa nhất định:

Tìm kiếm

```
struct list* search(struct list *L, keyType key)
{ while (L != NULL) {
    if (L->key == key) return L;
    L = L->next;
}
return NULL;
}
```

Đặc điểm khi được đưa vào đầu danh sách là thứ tự của các phần tử trong đó đổi lặp với thứ tự nhận của chúng. Để có thể làm việc đầy đủ với các danh sách, chúng ta sẽ cần thêm ba thao tác nữa, mỗi thao tác chúng ta sẽ xem xét riêng:

2.4.3. Đưa vào sau một phần tử được chỉ định bởi một chỉ dẫn đã cho

Chúng ta định nghĩa hàm bao gồm là

`void insertAfter (struct list ** L, keyType key, data x).`

Con trỏ kép L chỉ ra một mục trong danh sách (đặc biệt đây có thể là mục đầu tiên), sau đó là một mục có khóa key và trường dữ liệu verb!data!. Thuật toán chèn như sau:

1) Một phần tử trống được tạo key và giá trị x:

Chèn phần tử

```
struct list *temp;
temp = (struct list *) malloc(sizeof(*temp));
temp->key = key;
temp->info = x;
```

2) Chuyển hướng con trỏ: Phần tử mới sẽ trỏ đến phần tử sau L và L sẽ trỏ đến phần tử mới (Hình 2.8):

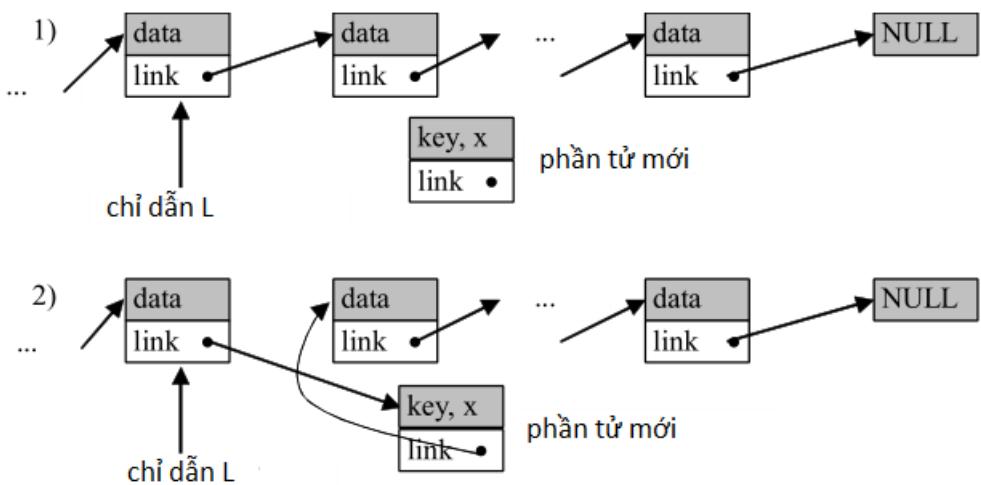
```
temp->next = (*L)->next;
(*L)->next = temp;
```

2.4.4. Đưa vào phía trước một phần tử được chỉ định bởi một thư mục

Hàm kích hoạt là

`void insertBefore(struct list **L, keyType key, data x).`

Việc đưa vào trước một phần tử được chỉ ra bởi L là phức tạp bởi thực tế là chúng ta không có quyền truy cập trực tiếp vào phần tử trước đó (để thay đổi con trỏ của nó next). Thủ thuật sau được sử dụng để giải quyết vấn đề: phần tử mới được chèn vào sau L theo cách đã mô tả ở trên, sau đó các giá trị tương ứng của các trường



Hình 2.8. Đưa vào sau phần tử chỉ ra từ bảng chỉ dẫn

của nó được trao đổi với giá trị của L.

```
struct list *temp;
temp = (struct list *) malloc(sizeof(*temp));
*temp = **L;
(*L)->next = temp;
(*L)->key = key;
(*L)->info = x;
```

2.4.5. Xóa theo khóa mặc định và con trỏ lên đầu danh sách

Chúng ta định nghĩa hàm xóa là

```
void deleteNode(struct list **L, keyType key)
```

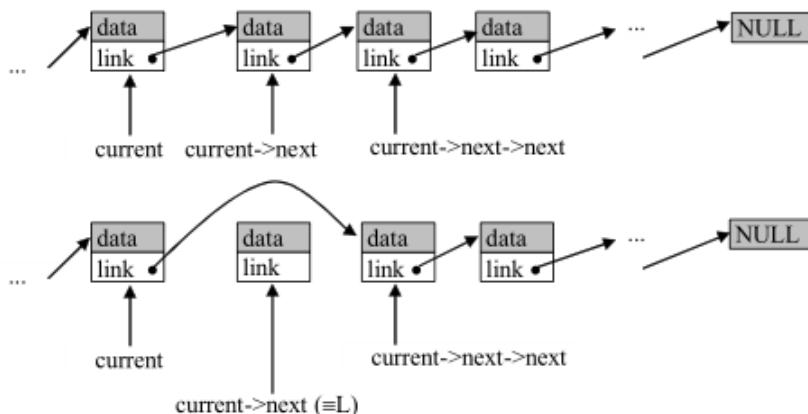
Bước đầu tiên là tìm yếu tố chính. Việc tìm kiếm được thực hiện theo cách mà tại thời điểm chúng ta tìm thấy mục cần xóa, chúng ta cũng có một con trỏ đến mục tiền nhiệm của nó:

```
(1) struct list *current = *L;
    while (current->next != NULL && current->next->key != key)
        current = current->next;
```

Do đó, phần tử xóa được chỉ định bởi current-> next. Quá trình xóa

được thực hiện như sau: Phần tử tiền nhiệm của phần tử xóa được đặt để trỏ đến phần tử kế nhiệm của nó, tức là phần tử bị "bỏ qua", sau đó bộ nhớ mà nó chiếm được giải phóng - Hình 2.9:

```
(2) save = current->next;
    current->next = current->next->next;
    free (save);
```



Hình 2.9. Loại trừ khỏi danh sách.

Khi loại trừ một mục khỏi danh sách, cần lưu ý một điều nữa - khi tìm kiếm (1), nếu đến cuối danh sách, nghĩa là không có mục nào có khóa đã cho và thông báo lỗi được hiển thị.

2.4.6. Xóa một phần tử được chỉ định bởi một thư mục

Thao tác xóa phần tử được chỉ ra bởi con trỏ L cũng tương tự như thao tác xóa bằng khóa - cần tìm phần tử trước của phần tử được chỉ thị bởi L và con trỏ của nó tới phần tử tiếp theo để được chuyển hướng một cách thích hợp. Do đó, độ phức tạp của việc xóa theo khóa, cũng như theo con trỏ tới một phần tử, là tuyến tính. Để đạt được độ phức tạp không đổi khi xóa theo thư mục, cần sử dụng danh sách tuyến tính hai liên kết (xem bài toán 2.4.). Sau đây là cách triển khai động đầy đủ của một danh sách được liên kết:

Chương trình 2.3. Thao tác trên danh sách (204list.c)

```

#include <stdio.h>
#include <stdlib.h>
typedef int data;
typedef int keyType;

struct list {
keyType key;
data info;
struct list *next;
};

/* đưa vào một phần tử ở đầu danh sách được liên kết*/
void insertBegin(struct list **L, keyType key, data x)
{
    struct list *temp;
    temp = (struct list *) malloc(sizeof(*temp));
    if (NULL == temp) {
        fprintf(stderr, " Không đủ bộ nhớ cho phần tử mới!\n");
        return;
    }
    temp->next = *L;
    (*L) = temp;
    (*L)->key = key;
    (*L)->info = x;
}

/* Đưa vào sau phần tử */
void insertAfter(struct list **L, keyType key, data x)
{ struct list *temp;
    if (NULL == *L) { /*nếu danh sách rỗng => trường hợp đặc biệt*/
        insertBegin(L, key, x);
        return;
    }

    temp = (struct list *) malloc(sizeof(*temp));
    /*Tạo ra phần tử mới */
    if (NULL == temp) {
        fprintf(stderr, " Không đủ bộ nhớ cho phần tử mới!\n");
        return;
}

```

```

    }
    temp->key = key;
    temp->info = x;
    temp->next = (*L)->next;
    (*L)->next = temp;
}

/* Đưa vào trước một phần tử */
void insertBefore(struct list **L, keyType key, data x)
{ struct list *temp;
if (NULL == *L) /*phần tử phải được chèn trước mục đầu tiên*/
    insertBegin(L, key, x);
    return;
}

temp = (struct list *) malloc(sizeof(*temp));
/* Tạo ra phần tử mới */
if (NULL == temp) {
    fprintf(stderr, "Không đủ bộ nhớ cho phần tử mới!\n");
    return;
}
*temp = **L;
(*L)->next = temp;
(*L)->key = key;
(*L)->info = x;
}

/*xóa một phần tử khỏi danh sách*/
void deleteNode(struct list **L, keyType key)
{ struct list *current = *L;
struct list *save;
if ((*L)->key == key) { /* mục đầu tiên phải được xóa */
    current = (*L)->next;
    free(*L);
    (*L) = current;
    return;
}
/*tìm phần tử cần xóa*/
while (current->next != NULL && current->next->key != key) {
    current = current->next;
}

```

```

    }
    if (NULL == current->next) {
        fprintf(stderr, "Lỗi: Không tìm thấy mục cần xóa! \n");
        return;
    }
    else {
        save = current->next;
        current->next = current->next->next;
        free(save);
    }
}
/* in các phần tử của một danh sách được liên kết */
void print(struct list *L)
{
    while (NULL != L) {
        printf("%d(%d) ", L->key, L->info);
        L = L->next;
    }
    printf("\n");
}
/* tìm kiếm một mục quan trọng trong danh sách được liên kết */
struct list* search(struct list *L, keyType key)
{
    while (L != NULL) {
        if (L->key == key) return L;
        L = L->next;
    }
    return NULL;
}

int main()
{
    struct list *L = NULL;
    int i, edata;
    insertBegin(&L, 0, 42);
    for (i = 1; i < 6; i++) {
        edata = rand() % 100;
        printf("Chèn trước: %d(%d)\n", i, edata);
        insertBefore(&L, i, edata);
    }

    for (i = 6; i < 10; i++) {
        edata = rand() % 100;
    }
}

```

```

printf("Chèn sau: %d(%d)\n", i, edata);
insertAfter(&L, i, edata);
}
print(L);
deleteNode(&L, 9); print(L);
deleteNode(&L, 0); print(L);
deleteNode(&L, 3); print(L);
deleteNode(&L, 5); print(L);
deleteNode(&L, 5);
return 0;
}

```

Kết quả thực hiện chương trình:

```

Chèn trước: 1 (46)
Chèn trước: 2 (30)
Chèn trước: 3 (82)
Chèn trước: 4 (90)
Chèn trước: 5 (56)
Chèn sau: 6 (17)
Chèn sau: 7 (95)
Chèn sau: 8 (15)
Chèn sau: 9 (48)
5 (56) 9 (48) 8 (15) 7 (95) 6 (17) 4 (90) 3 (82) 2 (30) 1 (46) 0 (42)
5 (56) 8 (15) 7 (95) 6 (17) 4 (90) 3 (82) 2 (30) 1 (46) 0 (42)
5 (56) 8 (15) 7 (95) 6 (17) 4 (90) 3 (82) 2 (30) 1 (46)
5 (56) 8 (15) 7 (95) 6 (17) 4 (90) 2 (30) 1 (46)
8 (15) 7 (95) 6 (17) 4 (90) 2 (30) 1 (46)
Lỗi: Không tìm thấy phần tử xóa!

```

Bài tập

- ▷ 2.5. Làm lại các chương trình trên để các hàm không in ra thông báo lỗi, nhưng trả về giá trị Boolean nếu chúng thành công.
- ▷ 2.6. Cấu trúc dữ liệu ngăn xếp, hàng đợi, bộ bài, v.v. chúng có thể được coi là danh sách tuyến tính đặc biệt và cũng có thể được thực hiện động. Chúng ta cho phép người đọc, bằng cách sử dụng triển khai danh sách không liên kết tuyến tính ở trên, cố gắng sửa đổi danh sách này thành triển khai ngăn xếp và hàng đợi động. Điều

quan trọng là độ phức tạp của các hoạt động cơ bản (bật, tắt) vẫn không đổi.

► 2.7. Để thực hiện một chiến lược kết hợp để trình bày một ngăn xếp trong bộ nhớ, sử dụng một danh sách các mảng. Ban đầu, nó bắt đầu với một mảng duy nhất và khi nó đầy, một khối mới được cấp phát, khối này được kết nối với một con trỏ tới khối trước đó. Khi mảng được đưa vào cuối cùng bị làm trống, nó sẽ bị loại trừ khỏi danh sách. Ưu điểm và nhược điểm của cách tiếp cận được đề xuất là gì? Bạn có giới thiệu nó cho một hàng đợi không? Và cho bộ bài?

2.5. Cây nhị phân

Việc sử dụng danh sách tuyến tính cực kỳ kém hiệu quả đối với các tác vụ trong đó hoạt động tìm kiếm được thực hiện rất thường xuyên so với các tác vụ khác liên quan đến việc thay đổi danh sách: bao gồm, xóa, sắp xếp lại, v.v.

Như chúng ta đã thấy, trong việc triển khai động của một danh sách tuyến tính, việc bao gồm và loại trừ một phần tử là các phép toán nhanh (độ phức tạp của chúng là không đổi - $\Theta(1)$), trái ngược với tìm kiếm khóa, có độ phức tạp là $\Theta(n)$.

Trong thực tế tinh, sự phức tạp của bao gồm là $\Theta(1)$, và loại trừ, cũng như nhu cầu - $\Theta(n)$. Có thể lưu trữ các mục trong danh sách, được sắp xếp theo khóa của chúng (chuyển đổi tĩnh). Điều này sẽ làm giảm hiệu quả của hoạt động bao gồm (các phần tử sẽ cần được sắp xếp lại), nhưng sẽ cho phép áp dụng tìm kiếm nhị phân cổ điển (được thảo luận trong Chương 4 - xem 4.3.) Và giảm độ phức tạp của tìm kiếm xuống $\Theta(\log_2 n)$. được gọi là hàng đợi ưu tiên, xem bài toán 2.6.). Không thể áp dụng tìm kiếm nhị phân trong chuyển đổi động vì không có quyền truy cập trực tiếp vào phần tử thứ k trong danh sách.

Chúng ta sẽ xem xét cấu trúc dữ liệu *dạng cây* kết hợp lợi ích của chuyển đổi động với hiệu suất tìm kiếm tĩnh. Trong cấu trúc này, ba phép toán bật, tắt và chìa khóa trao tay trong trường hợp ở giữa có độ phức tạp theo lôgarit.

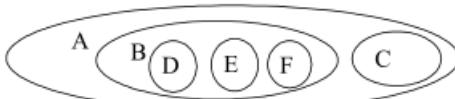
Định nghĩa 2.2. Tập hợp T là *một cây gốc* (sau đây chỉ gọi là cây) nếu

nó là một tập hợp rỗng hoặc nếu các điều kiện sau được thỏa mãn đồng thời:

- 1) Nó chứa một phần tử duy nhất t , được gọi là *đỉnh* của cây, mà chúng ta sẽ ký hiệu là $\text{root}(T)$.
- 2) Các phần tử khác (trừ gốc) được chia thành $m (m \geq 0)$ tập hợp rỗng không giao nhau T_1, T_2, \dots, T_m , mỗi tập là *một cây*.

Các tập T_1, T_2, \dots, T_m được gọi là *con* của T . Các gốc T_1 của T_1, T_2 của T_2, \dots, T_m của T_m được gọi là *con trực tiếp* của t . Các đỉnh khác của T_1, T_2, \dots, T_m là *người thừa kế gián tiếp* của t .

Kết quả là, mỗi phần tử (sau đây, các phần tử của T sẽ được gọi là *ngọn cây*) là gốc của một cây con. Số người thừa kế trực tiếp của một đỉnh là *mức độ* của nó. Nếu bậc của đỉnh bằng 0, nó được gọi là *lá*.



(A (B(D)(E)(F)) (C))

Hình 2.10. Biểu diễn trực quan thông qua các tập hợp lồng nhau.

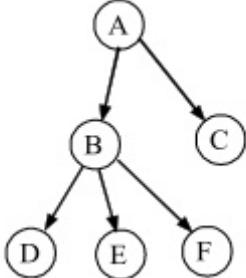
Hình 2.11. Trình bày trong ngoặc đơn.

Trong các Hình 2.10, 2.11, 2.12, 2.13 và ?? các cách trình bày trực quan khác nhau được thể hiện. Xét cây $T = \{A, B, C, D, E, F\}$ được thể hiện trong các hình sau: ở đây đỉnh A là gốc và $T_1 = \{C\}$ và $T_2 = \{B, D, E, F\}$ là các cây con của nó (đỉnh C là một lá). Trong cây $T_2 = \{B, D, E, F\}$ gốc là ngọn B , và các cây con là $\{D\}, \{E\}, \{F\}$ (D, E và F là lá).

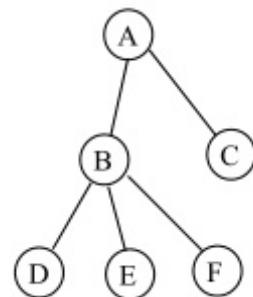
Định nghĩa 2.3. Đường đi trong cây là một dãy các đỉnh t_1, t_2, \dots, t_k không lặp lại và cứ hai đỉnh liên tiếp t_{i-1} và t_i ($2 \leq i \leq k$) thì thỏa mãn đúng một trong hai điều: t_i là *sự kế thừa* của t_{i-1} , hoặc t_{i-1} là *sự kế vị* của t_i . Số $k - 1$ được gọi là *độ dài* đường đi.

Mệnh đề 2.1. Chỉ có một lối đi giữa hai ngọn cây.

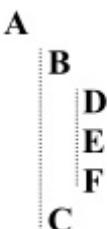
Định nghĩa 2.4. Mức đỉnh được gọi là *độ dài* của đường đi từ gốc đến ngọn. Chiều cao của cây được gọi là *mức tối đa* của đỉnh trong đó.



Hình 2.12. Biểu diễn bằng đồ thị liên thông xoay chiều có định hướng (xem Chương 5).



Hình 2.13. Biểu diễn bằng một đồ thị liên thông xoay chiều không có hướng (xem Chương 5), với một trong các đỉnh được chọn làm gốc: trong trường hợp này là đỉnh A.

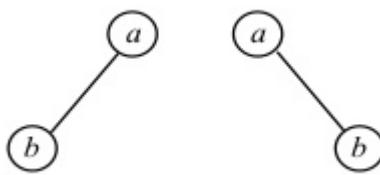


Hình 2.14. Biểu diễn bằng phép dời hình từng bước.

Định nghĩa 2.5. Nếu độ của mỗi ngọn cây nhỏ hơn hoặc bằng hai thì cây đó được gọi là *cây nhị phân*. Vì có nhiều nhất hai cây con trên mỗi đỉnh, nên một sắc lệnh của những người thừa kế (và của các cây con, tương ứng) thường được đưa ra nhiều nhất: *trái* và *phải*.

Cần lưu ý rằng, sau khi chúng ta đưa ra một quy định, về mặt cấu trúc, cây nhị phân khác về chất với cây và không phải là một tập con của chúng. Ví dụ, hai đối tượng trong Hình 2.15. chúng hoàn toàn không mô tả cùng một cây nhị phân, mặc dù là cây, chúng không thể phân biệt được và đại diện cho cây trong Hình 2.16.

Đến cuối đoạn này, chúng ta sẽ xem xét cây nhị phân. Thực hiện theo một số ví dụ:



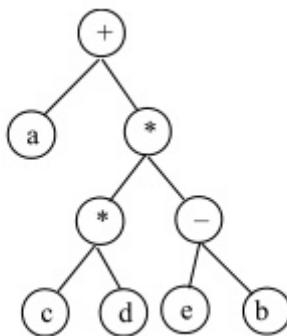
Hình 2.15. Cây nhị phân



Hình 2.16. Cây

- Giải đấu quần vợt, nơi nó được chơi với việc loại bỏ trực tiếp.
- Trình bày biểu thức số học với các giao dịch hai phần (xem Hình 2.17): Mỗi thao tác là một đỉnh với những người thừa kế cả hai toán hạng của nó.

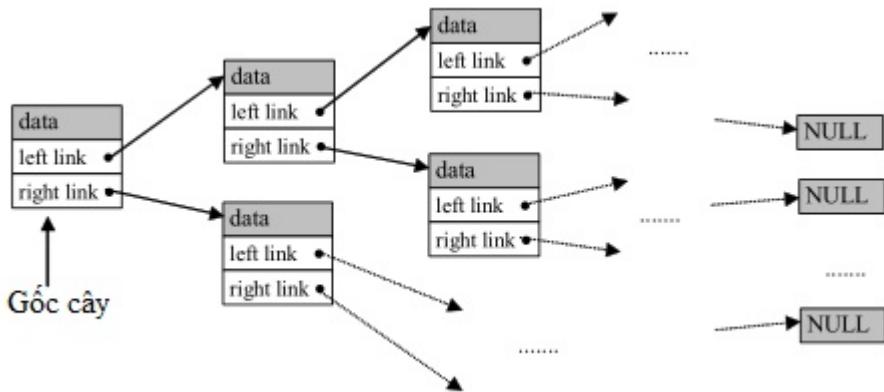
Cây nhị phân được sử dụng rộng rãi trong trình biên dịch, cơ sở dữ liệu và nhiều hơn nữa.

Hình 2.17. Trình bày biểu thức $a + ((c * d) * (e \tilde{v} b))$

Hãy là một cây nhị phân T , với các thông tin sau (Hình 2.18):

- Khóa khóa key(t) trên đầu trang.
- Trường data(t) với dữ liệu hàng đầu bổ sung.
- Hai thư mục left (t) và right (t): Ở bên trái và bên phải của T tương ứng của t

```
struct tree {
    keyType key;
    data info;
    struct tree * left ;
    struct tree * right;
};
```



Hình 2.18. Trình bày động của một cây nhị phân trong bộ nhớ.

Chúng ta sẽ xác định ba thao tác cơ bản trên cây nhị phân:

- Thêm một đỉnh

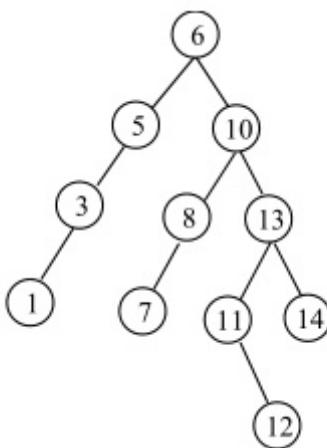

```
void insertKey(keyType key, data x, struct tree **T);
```
- Xóa một khóa chính


```
void deleteKey(keyType key, struct tree **T);
```
- Tìm kiếm một đỉnh (và truy xuất dữ liệu từ nó) bằng khóa


```
struct tree *search(keyType key, struct tree *T);
```

Nhiều loại cây nhị phân là *cây tìm kiếm nhị phân*. Với sự giúp đỡ của họ, chúng ta sẽ thực hiện các phép toán vừa liệt kê để độ phức tạp của chúng là logarit.

Định nghĩa 2.6. *Cây tìm kiếm nhị phân* (còn gọi là *cây nhị phân*) là một cây nhị phân cho mỗi đỉnh được thực thi: khóa của nó lớn hơn khóa của tất cả các phần tử trong cây con bên trái của nó và nhỏ hơn (hoặc bằng trong trường hợp cho phép trong cây. để thêm các phần tử có cùng khóa) từ các khóa của tất cả các phần tử ở bên phải.



Hình 2.19. Cây tìm kiếm nhị phân.

Hình 2.19 hiển thị một cây tìm kiếm nhị phân (các khóa phần tử là các số nguyên). Có thể thấy rằng khóa trên mỗi đỉnh có giá trị cao hơn các khóa trong cây con bên trái của nó và giá trị thấp hơn các khóa trong cây con bên phải của nó. Đặc biệt, điều kiện sau được đáp ứng cho những người thừa kế của mỗi đỉnh (điều này quan trọng trong việc thực hiện các hoạt động trong cây).

Tìm kiếm khóa trong cây nhị phân được thực hiện như sau:

2.5.1. Tìm kiếm bằng chìa khóa

Chúng ta bắt đầu tìm kiếm từ gốc, tức là $t = \text{root } (T)$;

$\text{search}(\text{key}, t)$:

1) Nếu $\text{key} < \text{key } (t)$, ta tiếp tục tìm kiếm trong cây con bên trái của t , tức là ta thực hiện tìm kiếm đệ quy $\text{search}(\text{left } (t), \text{key})$;

2) Nếu $\text{key} > \text{key } (t)$, ta tiếp tục tìm kiếm trong cây con bên phải của t , tức là ta thực hiện tìm kiếm đệ quy $\text{search}(\text{right } (t), \text{key})$;

3) Nếu $\text{key} == \text{key } (t)$, chúng ta đã tìm thấy đỉnh cần thiết.

Trong thuật toán trên, chúng ta đã giả định rằng khóa chúng ta đang tìm kiếm nằm trong cây. Nếu bạn tìm kiếm một khóa không tồn tại, thì đến một bước nào đó, cây cần tìm sẽ trống (tức là con trỏ cha tương ứng cho biết NULL). Ví dụ, nếu trong cây của Hình 2.19 chúng ta đang tìm kiếm một phần tử có khóa 9, thuật toán sẽ thực

hiện liên tiếp các kiểm tra sau:

$9 > 6$ - chuyển đến gốc của cây con bên phải - trên cùng với khóa 10.

$9 < 10$ - chuyển đến gốc của cây con bên trái - trên cùng với khóa 8.

$9 > 8$ - chúng ta phải đi đến gốc của cây con bên phải, nhưng nó trả đến NULL, vì vậy phần tử được tìm kiếm bị thiếu.

2.5.2. Thêm vào một đỉnh mới

Việc đưa vào được thực hiện tương tự như tìm kiếm, mục tiêu ở đây là đến được một cây trống để có thể thêm một đỉnh mới vào đó.

insert (t, p):

1) Nếu $t == \text{NULL}$ thì sau đó chúng ta đã tìm thấy nơi đưa đỉnh mới p và đưa nó vào: $t = p$;

2) Nếu $\text{key}(p) < \text{key}(t)$ insert (left (t), key);

3) Nếu $\text{key}(p) > \text{key}(t)$ insert (right (t), key);

4) Nếu $\text{key}(p) == \text{key}(t)$ thì phần tử cần chèn đã có trong cây.

Trong trường hợp này, chúng ta có hai lựa chọn: chúng ta không thể làm gì, hoặc chúng ta có thể hiển thị thông báo lỗi (tức là chúng ta không cho phép các khóa trùng lặp).

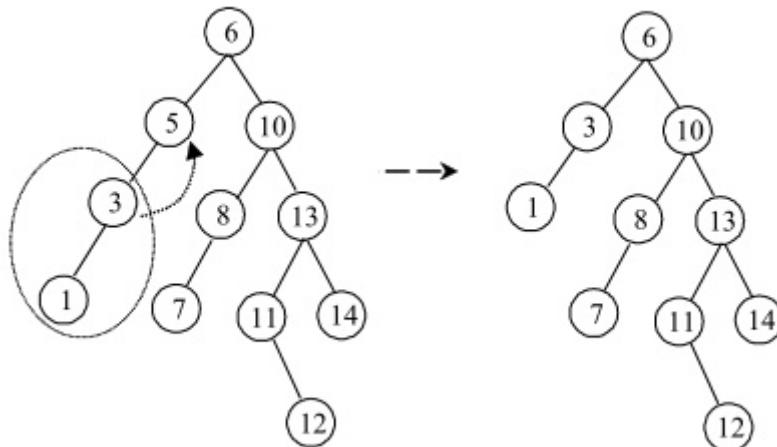
2.5.3. Xóa một đỉnh bằng từ khóa đã cho

Thao tác này phức tạp hơn một chút. Rõ ràng là trước khi xóa mèo có khóa k khỏi cây, chúng ta cần tìm nó. Điều này được thực hiện theo thuật toán đã được mô tả để tìm một đỉnh. Sau đó, 3 tình huống có thể xảy ra:

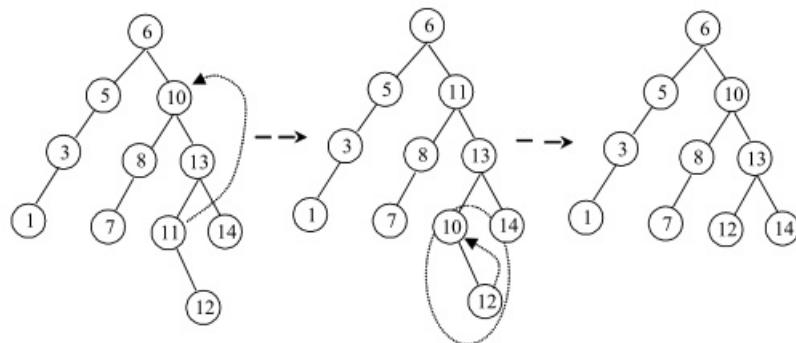
- Nếu đỉnh là một lá - vùng nhớ bị chiếm bởi nó được giải phóng và con trỏ của đỉnh trỏ đến nó bị thay đổi (nó được gán giá trị NULL).
- Nếu trên cùng chỉ có cây con bên trái hoặc chỉ cây con bên phải thì thay bằng gốc của cây con đó.
- Trường hợp phức tạp nhất là khi mũi xóa p có cả cây con trái và phải. Sau đó áp dụng cách sau: Tìm đỉnh có khóa nhỏ nhất trong cây con bên phải (ngoài cùng bên trái trong cây con bên phải) và đổi đỉnh này với p . Sau khi trao đổi p sẽ có nhiều nhất một cây con và bị loại trừ bởi một trong hai quy tắc trên. (Tất

nhiên, chúng ta có thể hoán đổi trái với phải và ngược lại: tìm phần tử có khóa lớn nhất trong cây con bên trái, hoán đổi nó với p , v.v.)

Hai trường hợp cuối cùng được minh họa trong Hình 2.20 và Hình 2.21.



Hình 2.20. Xóa kết nối đầu bằng từ khóa 5.



Hình 2.21. Xóa kết nối đầu bằng từ khóa 10.

Sau đây là cách triển khai đầy đủ các hoạt động được mô tả ở trên:

Chương trình 2.4. Thao tác trên cây nhị phân (205bintree.c)

```

#include <stdio.h>
#include <stdlib.h>

typedef char *data;
typedef int keyType;
struct tree {
    keyType key;
    data info;
    struct tree * left ;
    struct tree * right ;
};

/*Tìm kiếm trong cây nhị phân*/
struct tree *search(keyType key, struct tree *T)
{ if (NULL == T)
    return NULL;
else if (key < T->key)
    return search(key, T->left);
else if (key > T->key)
    return search(key, T->right);
else
    return T;
}

/*Đưa vào cây nhị phân*/
void insertKey(keyType key, data x, struct tree **T)
{ if (NULL == *T) {
    *T = (struct tree *) malloc(sizeof(**T));
    (*T)->key = key;
    (*T)->info = x;
    (*T)->left = NULL;
    (*T)->right = NULL;
}
else if (key < (*T)->key)
    insertKey(key, x, &(*T)->left);
else if (key > (*T)->key)
    insertKey(key, x, &(*T)->right);
else
    fprintf(stderr, "Phản tử đã có trong cây!\n");
}

```

```
/*Loại trừ khỏi cây nhị phân*/
/* Tìm phần tử tối thiểu trong cây*/
struct tree *findMin(struct tree *T)
{ while (NULL != T->left) T = T->left;
  return T;
}
void deleteKey(keyType key, struct tree **T)
{ if (NULL == *T) {
    fprintf(stderr,"Đỉnh cần xóa không có!\n");
} else {
    if (key < (*T)->key)
        deleteKey(key, &(*T)->left);
    else if (key > (*T)->key)
        deleteKey(key, &(*T)->right);
    else /* phần tử loại trừ đã được tìm thấy*/
        if ((*T)->left && (*T)->right) {/*đỉnh có hai nhánh thừa kế*/
            /*có một đỉnh để trao đổi */
            struct tree *replace = findMin((*T)->right);
            (*T)->key = replace->key;
            (*T)->info = replace->info;
            deleteKey((*T)->key, &(*T)->right); /*Đỉnh loại trừ */
        }
        else /* phần tử không có hoặc một cây con */
        {
            struct tree *temp = *T;
            if ((*T)->left)
                *T = (*T)->left;
            else
                *T = (*T)->right;
            free(temp);
        }
    }
}
void printTree(struct tree *T)
{ if (NULL == T) return;
  printf("%d ", T->key);
  printTree(T->left);
  printTree(T->right);
}
```

```

int main() {
    struct tree *T = NULL, *result;
    int i;
    /* đưa vào đỉnh 10 với các khóa ngẫu nhiên */
    for (i = 0; i < 10; i++) {
        int ikey = (rand() % 20) + 1;
        printf("Chèn phần tử bằng khóa %d \n", ikey);
        insertKey(ikey, "someinfo", &T);
    }
    printf("Cây: ");
    printTree(T);
    printf("\n");
    /* tìm kiếm phần tử bằng khóa 5 */
    result = search(5, T);
    printf("đã tìm thấy: %s\n", result->info);

    /* xóa đỉnh 10 ngẫu nhiên khỏi cây */
    for (i = 0; i < 10; i++) {
        int ikey = (rand() % 20) + 1;
        printf("Xóa phần tử bằng khóa %d \n", ikey);
        deleteKey(ikey, &T);
    }
    printf("Cây: ");
    printTree(T);
    printf("\n");
    return 0;
}

```

Kết quả thực hiện chương trình:

Chèn phần tử bằng khóa 1
 Chèn phần tử bằng khóa 1
 Phần tử đã có trong cây!
 Một phần tử có khóa 7 được chèn
 Chèn phần tử bằng khóa 1
 Phần tử đã có trong cây!
 Một phần tử có khóa 8 được chèn
 Chèn phần tử bằng khóa 5
 Một phần tử có khóa 11 được chèn
 Chèn phần tử bằng khóa 4

Chèn phần tử bằng khóa 15
 Chèn phần tử bằng khóa 19
 Cây: 1 7 5 4 8 11 15 19
 Tìm thấy: someinfo
 Xóa phần tử bằng khóa 6
 Đỉnh cần xóa không có!
 Xóa phần tử bằng khóa 9
 Đỉnh cần xóa không có!
 Xóa phần tử bằng khóa 3
 Đỉnh cần xóa không có!
 Xóa phần tử bằng khóa 14
 Đỉnh cần xóa không có!
 Xóa phần tử bằng khóa 12
 Đỉnh cần xóa không có!
 Xóa phần tử bằng khóa 1
 Xóa phần tử bằng khóa 4
 Xóa phần tử bằng khóa 17
 Đỉnh cần xóa không có!
 Xóa phần tử bằng khóa 14
 Đỉnh cần xóa không có!
 Xóa phần tử bằng khóa 16
 Đỉnh cần xóa không có!
 Cây: 7 5 8 11 15 19

2.5.4. Thu thập thông tin

Có thể thấy rằng trong chương trình trên chúng ta cũng đã sử dụng một hàm để in ra tất cả các đỉnh trên cây nhị phân printTree (**struct** tree * T). Hàm là đệ quy và dựa trên thuật toán sau: Phần gốc của cây được in, sau đó (đệ quy!) phần bên trái được in, và sau đó là cây con bên phải của T.

Không đưa ra định nghĩa chính thức (sẽ có một định nghĩa khi xem xét các cột trong Chương 5), dưới bò một cái cây, chúng ta sẽ hiểu việc kiểm tra tuần tự từng ngọn của nó, bắt đầu từ gốc.

Chức năng trên để in cây thực tế là rùa bò. Trong đó, gốc được kiểm tra đầu tiên, và chỉ sau đó đến ngọn của các cây con bên trái và bên phải. Thu thập thông tin này được gọi là *thu thập thông tin gốc*-

trái-phải hoặc preorder: CLD. Rõ ràng, sử dụng cùng một nguyên tắc, cây có thể được thu thập thông tin theo hai cách khác nhau đáng kể:

- LCD (Inorder): Được xem theo thứ tự: cây con bên trái, cây con gốc và cây con bên phải.
- LDC (Postorder): Các ngọn của cây con bên trái và bên phải được kiểm tra đầu tiên, và chỉ sau đó mới kiểm tra gốc.

Mỗi loại trong ba loại thu thập thông tin được sử dụng trong các bài toán và cách diễn giải khác nhau của cây nhị phân. Ví dụ, nếu chúng ta áp dụng một inorder cho cây tìm kiếm nhị phân và in khóa trên mỗi đỉnh mà chúng ta truy cập, chúng ta sẽ nhận được các khóa được sắp xếp theo thứ tự tăng dần.

Hãy xem xét cây nhị phân trong Hình 2.18. Hãy đi xung quanh nó và xem điều gì sẽ xảy ra:

- CLD: + a ** cd - eb
- LCD: a + c * d * e - b
- LDC: acd * eb - * +

Mỗi loại trong ba kiểu bò đều có kiểu "gương" riêng - chúng ta trao đổi thứ tự bò trên cây con bên trái và bên phải:

- CDL: + * - be * dca
- DCL: b - e * d * c + a
- DLC: be - dc ** a +

Như vậy, tổng số các kiểu duyệt sẽ trở thành 6: nhiều như hoán vị của các chữ cái L (bên trái), D (bên phải) và C (gốc). Có thể thấy rằng ký hiệu đại số tiêu chuẩn của biểu thức có được bằng cách thu thập dữ liệu kiểu LCD (tuy nhiên, không giữ lại các ưu tiên trong tính toán và để được sử dụng trong thực tế, phải được sửa đổi theo cách thích hợp - làm thế nào?), và trong CLD và LDC thu được bản ghi trường trực tiếp và đảo ngược tương ứng. Ký hiệu trường thẳng và đảo ngược rất hữu ích vì chúng có thể dễ dàng được sử dụng để tính toán một giá trị số mà không cần sử dụng dấu ngoặc đơn để thay đổi mức độ ưu tiên của các phép toán (khi so sánh giá trị của các biến trong một biểu thức, trong trường hợp này là a, c, d, e và f) [Wirth-1980] [Nakov-1998].

2.5.5. Bài tập

► 2.8. Xây dựng cây tìm kiếm nhị phân có thứ tự bằng cách thêm tuần tự các đỉnh sau:

- a) 7, 14, 28, 35, 65, 12, 18, 42, 81, 64, 61, 4, 13
- b) 12, 7, 14, 81, 42, 18, 61, 4, 64, 35, 13, 28, 65
- c) 4, 7, 12, 13, 14, 18, 28, 35, 42, 61, 64, 65, 81
- e) 81, 65, 64, 61, 42, 35, 28, 18, 14, 13, 12, 7, 4
- e) 28, 64, 13, 42, 7, 81, 61, 4, 12, 65, 35, 18, 14

So sánh các cây kết quả. Có thể rút ra kết luận gì?

► 2.9. 2. Theo thứ tự này, xóa các ngọn 8, 13, 5 và 6 của cây trong Hình 2.19.

► 2.10. Biểu thức $(A + B - C) * (D/F) - G * H$ đã cho. Tìm bản ghi tiếng Ba Lan ngược của anh ấy.

► 2.11. Chứng minh rằng chỉ có một đường đi giữa hai đỉnh của cây.

► 2.12. Phần nhỏ nhất của "cây" trong Hình 2.17 là gì. nên được cắt bớt để ví dụ đáp ứng Định nghĩa 2.2.

► 2.13. Viết phiên bản lặp lại của hàm thu thập thông tin cây nhị phân.

► 2.14. Viết một biến thể của hàm để duyệt cây nhị phân, hàm này sẽ in ra để có thể nhìn thấy cấu trúc của nó.

► 2.15. Để chứng minh tính đúng đắn của thuật toán đã mô tả để xóa một đỉnh khỏi cây nhị phân có thứ tự.

► 2.16. Thuật toán được mô tả ở trên để xóa một đỉnh khỏi cây nhị phân có thứ tự trong trường hợp hai người thừa kế đang tìm kiếm người thừa kế ngoài cùng bên trái của cây con bên phải. Triển khai một tùy chọn tìm kiếm cây kế thừa ngoài cùng bên phải cho cây con bên trái. Có khả năng nào khác không?

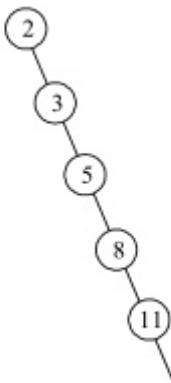
► 2.17. Đề xuất và thực hiện một biểu diễn tinh của một cây nhị phân.

► 2.18. Ở trên chúng ta đã thấy rằng cách chuẩn để biểu diễn một biểu thức trong cây nhị phân là viết các biến và hằng số trong danh sách và các phép toán trong các đỉnh. Lưu ý rằng chúng ta chỉ cho phép các phép toán hai đối số (chẳng hạn như $+$, $-$, $*$, $/$), với người thừa kế bên trái chứa đối số đầu tiên và người thừa kế bên phải là đối số thứ hai. Có thể cho phép các phép toán một ngôi (đối số đơn) như: $+$ và $-$ không? Điều gì đang thay đổi?

2.6. Cây cân đối

Phân tích kỹ hơn các phép toán với cây có thứ tự nhị phân cho thấy rằng có những trường hợp cây kết quả phân nhánh rất yếu và giống như một danh sách trong cấu trúc, và do đó về hiệu quả. Hãy để chúng ta, bắt đầu từ một cây trống, thực hiện một chuỗi gồm n phần tử. Kết quả thu được có thể cực kỳ "khó chịu" - ví dụ, nếu số lượng khóa của các phần tử được bao gồm tăng lên nghiêm ngặt (Hình ??):

$$2, 3, 5, 8, 11, \dots$$



Hình 2.22. Trường hợp xấu nhất với cây nhị phân.

Cây kết quả "suy biến" thành một danh sách liên kết, tìm kiếm mà, như chúng ta biết, có độ phức tạp tuyến tính. Thông thường trong thực tế, chúng ta cần tìm kiếm nhanh, bất kể loại đầu vào là gì, và trong những trường hợp như vậy, độ phức tạp tuyến tính

là một giải pháp không thể chấp nhận được trong trường hợp xấu nhất.

Chúng ta sẽ xem xét một cấu trúc dữ liệu "giữ" cây ở trạng thái trong đó các phép toán cơ bản (bật, tắt, tìm kiếm) có độ phức tạp logarit ngay cả trong trường hợp xấu nhất.

Định nghĩa 2.7. Cây nhị phân được gọi là *cân bằng* nếu sự khác biệt về chiều cao của các cây con bên trái và bên phải ở mỗi đỉnh nhiều nhất là một.

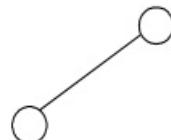
Định nghĩa 2.8. Một cây nhị phân có n đỉnh được gọi là *cân bằng hoàn toàn* nếu sự khác biệt về số lượng đỉnh của cây con bên trái và bên phải của mỗi đỉnh nhiều nhất là một.

Định nghĩa 2.9. Định nghĩa 2.9. Cây Fibonacci T_k có thứ tự k được gọi là cây nhị phân trong đó:

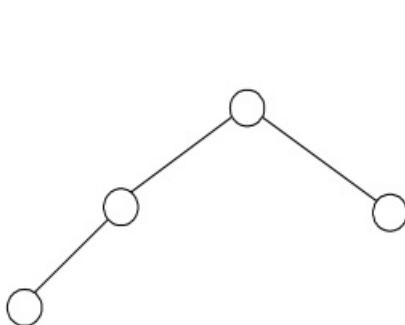
- T_0 là cây rỗng - có chiều cao 0;
- T_1 là cây chứa một nút đơn - có chiều cao 1;
- Với $k \geq 2$, cây bao gồm một gốc, một cây Fibonacci T_{k-1} của hàng $k-1$ (cây con bên trái) và một cây Fibonacci T_{k-2} của hàng $k-2$ (cây con bên phải).



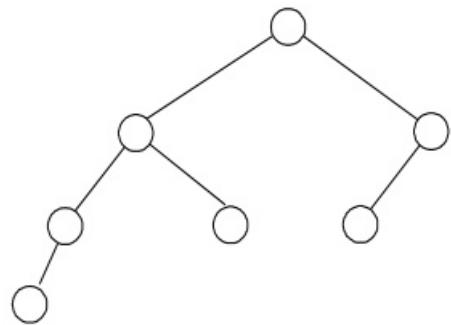
Hình 2.23. Cây Fibonacci của hàng 0.



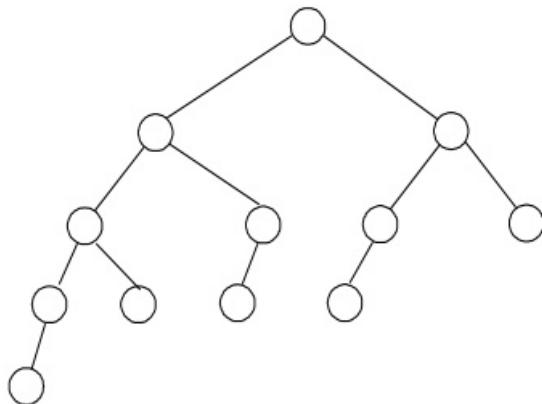
Hình 2.24. Cây Fibonacci của hàng 1.



Hình 2.25. Cây Fibonacci của hàng 2.



Hình 2.26. Cây Fibonacci của hàng 3.



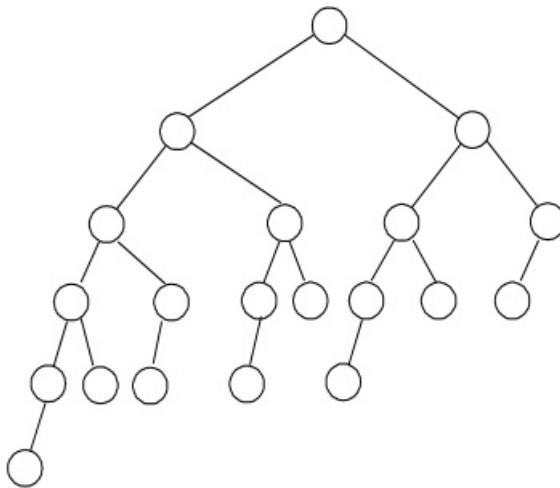
Hình 2.27. Cây Fibonacci của hàng 4.

Một vài cây Fibonacci đầu tiên được thể hiện trong Hình 2.23, 2.24, 2.25, 2.26, 2.27. và 2.28. Dễ dàng nhận thấy rằng cây Fibonacci là trường hợp xấu nhất mà chúng ta có thể mắc phải đối với cây cân bằng nhị phân. Thực tế này cũng được sử dụng trong định lý sau:

Định lý 2.1 (Adelson, Welsh và Landis). *Cho là một cây nhị phân T với n đỉnh bên trong. Gọi h là chiều cao của T . Khi đó*

$$\log_2(n + 1) < h < h1,4404\log_2(n + 2) - 0,3277.$$

Kết quả cuối cùng thu được như thế nào? Rõ ràng, cây nhị phân có chiều cao h không thể có nhiều hơn 2^h đỉnh bên trong, tức là $n + 1 \leq 2^h$, hoặc $h \geq \lceil \log_2(n + 1) \rceil$.



Hình 2.28. Cây Fibonacci của hàng 5.

Để giới hạn h từ phía trên, chúng ta hãy ký hiệu T_h là một cây cân bằng, với tính chất sau: T_h có chiều cao h và có số đỉnh tối thiểu. Vì T_h là cân bằng, (không giới hạn cộng đồng) cây con bên trái của T_h là $h - 1$ và cây con bên phải là $h - 1$ hoặc $h - 2$. Vì T_h là cây có số đỉnh tối thiểu nên cây con bên trái của gốc của T_h sẽ là T_{h-1} và bên phải: T_{h-2} . Trong sự tiếp tục quy nạp của việc xây dựng, chúng ta sẽ nhận được rằng T_h sẽ là cây Fibonacci bậc $h + 1$ (Tại sao?). Do đó $n \geq F_h + 2$, và từ thuộc tính [Knuth-3/1968]:

$$F_{h+2} - 1 > \varphi^{h+2} / \sqrt{5} - 2.$$

phần khác của các bất đẳng thức thu được.

Bây giờ trở lại cây nhị phân để tìm kiếm. Khi liên tiếp thêm các phần tử mới, về cơ bản có hai cách tiếp cận để giữ cây tìm kiếm theo cách "cân bằng":

- tái cấu trúc cây nhị phân trong quá trình xây dựng.
- giảm cấp độ bằng cách sử dụng các cây có bậc cao hơn - hàng thứ k .

Hai cách tiếp cận này, cũng như một số giống cây cân đối cụ thể, sẽ được thảo luận trong hai đoạn tiếp theo.

Bài tập

► 2.19. Xây dựng cây tìm kiếm nhị phân cân bằng hoàn hảo cho tập hợp sau các đỉnh $\{4, 7, 12, 13, 14, 18, 28, 35, 42, 61, 64, 65, 81\}$.

► 2.20. Cây có cân đối không từ:

- a) Hình 2.13.
- b) Hình 2.18.
- c) Hình 2.19.

► 2.21. Cây có cân đối hoàn hảo từ:

- a) Hình 2.13.
- b) Hình 2.18.
- c) Hình 2.19.

2.6.1. Vòng xoay. Cây đỏ và đen

Để duy trì chiều cao logarit của cây tìm kiếm nhị phân, các phép tái cấu trúc đặc biệt (phép quay) được xác định, được áp dụng bất cứ khi nào xảy ra "sự mất cân bằng". Đây là xoay trái và phải, xoay đôi và những thứ khác. (Hình 2.29).

Các thuộc tính quan trọng của các phép quay được hiển thị là:

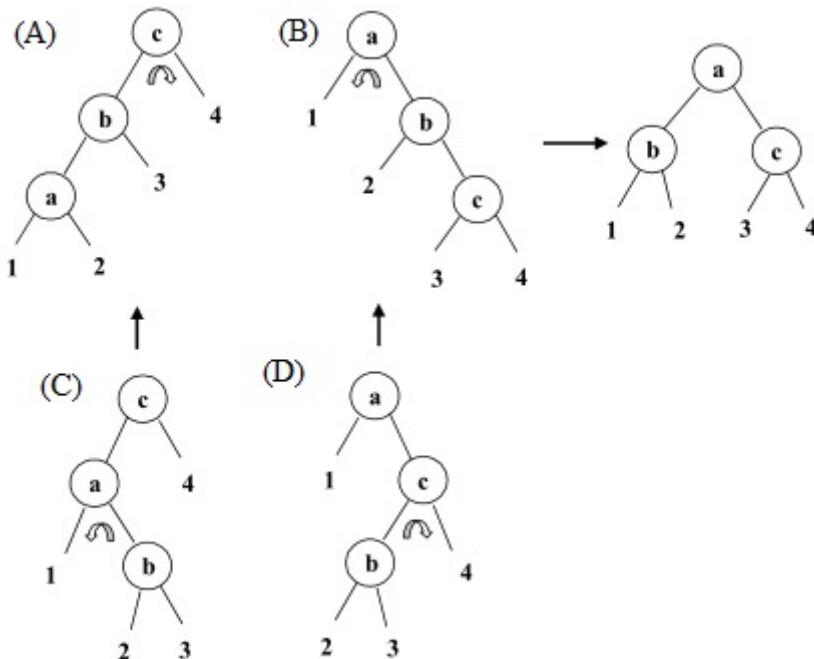
- việc thực hiện chúng vừa đơn giản vừa hiệu quả;
- bảo toàn kết quả thu thập thông tin cây kiểu LCD;
- đảm bảo cân bằng (nhưng không phải là cân bằng lý tưởng) của cây.

Việc lựa chọn phép quay nào để thực hiện trên các đỉnh khác nhau phải được làm rõ thêm liên quan đến các thuật toán cụ thể. Như một ví dụ, chúng ta sẽ xem xét một cấu trúc dữ liệu thú vị và phổ biến - cái gọi là Cây đỏ-đen.

Định nghĩa 2.10. Cây đỏ-đen được gọi là cây tìm kiếm nhị phân, trong đó mỗi đỉnh được đánh dấu đỏ hoặc đen và các đặc tính sau được bổ sung:

- Tất cả các lá (đỉnh NULL, chúng không thực sự có mặt) đều có màu đen.
- Nếu một đỉnh có màu đỏ thì hai đỉnh kế tiếp của nó có màu đen.

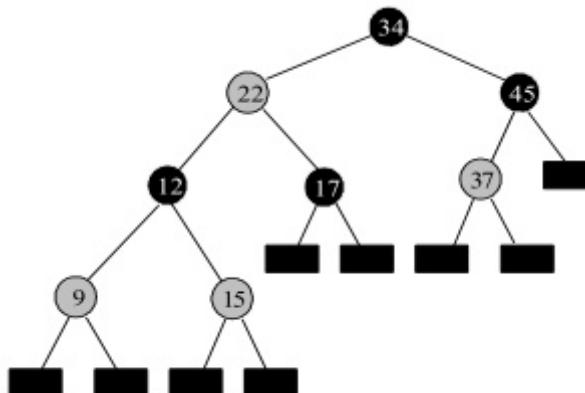
- Tất cả các đường đi từ đỉnh t bất kỳ đến bất kỳ lá nào từ cây con có gốc t đều chứa cùng một số đỉnh đen.



Hình 2.29. Xoay phải (A), trái (B), xoay trái-phải (C) và xoay phải-trái (D) quay.

Hệ quả trực tiếp của Định nghĩa 2.10 là tính chất hữu ích sau: chiều cao của cây đỏ đen với n đỉnh nhiều nhất là $2 \log_2(n + 1)$, tức là cây đỏ đen là cây tìm kiếm nhị phân xấp xỉ cân bằng. Tuy nhiên, chúng không cân bằng theo định nghĩa của chúng ta. Nhớ lại rằng một cây là cân bằng, mà sự khác biệt về chiều cao của cây con bên trái và bên phải của mỗi đỉnh cao nhất là 1. Trong màu đỏ-đen, sự khác biệt này nhiều nhất là hai lần. Đổi với cây đỏ đen, có các thuật toán tương đối đơn giản cho các phép toán cơ bản (bắt, tìm kiếm, tắt) với độ phức tạp $\Theta(\log_2 n)$, đó là lý do tại sao chúng được sử dụng rộng rãi nhất trong thực tế cây cân bằng [Shishkov-1995] [Cormen, Leiserson, Rivest-1997].

Có những loại cây nhị phân tìm kiếm cân bằng cổ điển khác, chẳng hạn như cây AVL (được đặt theo tên của Adelson-Velskii và



Hình 2.30. Ví dụ về cây đỏ-đen.

Landis), cũng đảm bảo độ phức tạp logarit cho các phép toán trên [Wirt-1980] [Knuth-3/1968]. Tuy nhiên, các thuật toán làm việc với cây AVL phức tạp hơn và hiệu quả của chúng trong thực tế thường kém hơn so với màu đỏ và đen, đó là lý do tại sao chúng ít được sử dụng hơn.

Bài tập

- 2.22. Sử dụng Định nghĩa 2.10, Để chứng minh rằng mỗi cây rụng lá bằng cây đỏ - đen có đúng hai người thừa kế.
- 2.23. Để xem xét các thao tác chính để làm việc với cây đỏ và đen có thể được thực hiện như thế nào trong mô tả đã cho về cấu trúc của chúng.

2.6.2. B-Cây

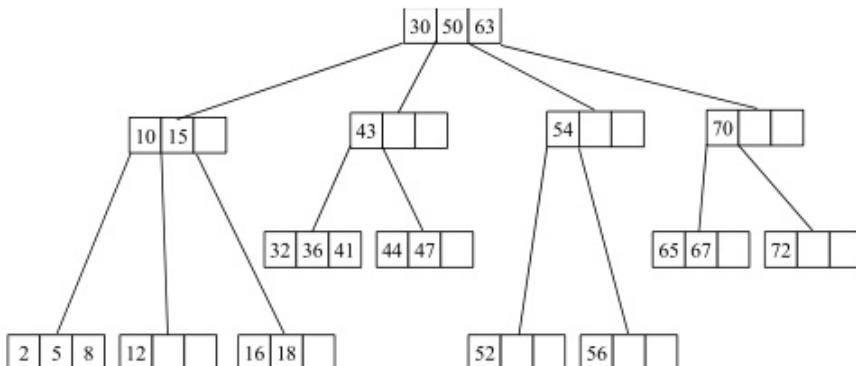
Ý tưởng xây dựng cây tìm kiếm (trong tài liệu cho đến nay chúng ta chỉ coi là cây nhị phân) có thể được mở rộng cho các cây ở mức độ cao hơn.

Định nghĩa 2.11. Chúng ta gọi *một đỉnh k* trong cây là một đỉnh chứa thông tin về $k - 1$ phần tử (ở đây một phần tử có nghĩa là một khóa + dữ liệu bổ sung) với các khóa $t_1 < t_2 < \dots < t_{k-1}$ và với k kẽ. - Cây con T_1, T_2, \dots, T_k , trong đó đối với mỗi phím $t_i, i = 1, 2, \dots, k - 1$,

điều kiện sau được đáp ứng: tất cả các phím trong cây con T_i đều nhỏ hơn t_i và tất cả các phím trong cây con T_{i+1} đều lớn hơn t_i .

Định nghĩa 2.12. Cây 2-3-4 được gọi là cây T với các đặc tính sau:

- T là cây hoàn toàn cân đối.
- Mỗi khối chót bên trong (không phải là chiếc lá) đều là khối chót 2, hoặc khối 3, hoặc khối 4.
- Tất cả các đường đi từ gốc đến bất kỳ lá nào đều có cùng chiều dài.



Hình 2.31. 2-3-4 cây.

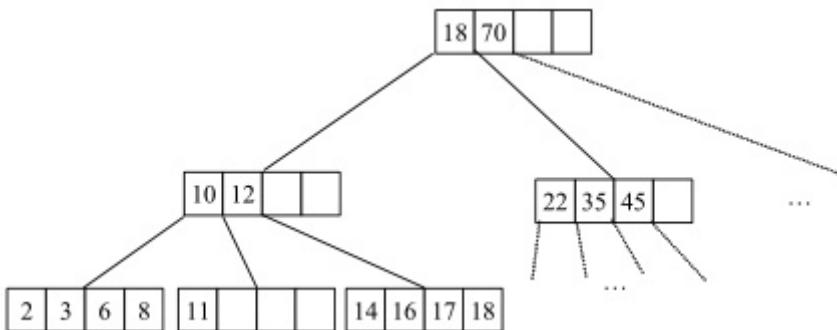
Có các thuật toán hiệu quả để làm việc với 2-3-4 cây, đảm bảo độ phức tạp của các phép toán cơ bản (thêm, tìm, xóa) bậc $\Theta(\log_2 n)$. Cũng cần lưu ý rằng có một cách để chuyển đổi cây 2-3-4 thành cây nhị phân tương đương (cụ thể là cây đố-đen đã thảo luận ở trên), cho phép chúng được trình bày đơn giản hơn trong bộ nhớ và tồn tại các thuật toán đơn giản hơn. Đó là lý do tại sao trong thực tế, cây đố-đen thường được sử dụng thay vì cây 2-3-4 cây [Flamig-1993].

Chúng ta sẽ xem xét một bản tóm tắt quan trọng của 2-3-4 cây.

Định nghĩa 2.13. B-cây của hàng m được gọi là cây tìm kiếm với các thuộc tính sau:

- Mỗi đỉnh, trừ gốc và lá, là đỉnh k , với k là một số trong khoảng từ $[m/2]$ đến m .
- Tất cả các đường đi từ gốc đến bất kỳ lá nào đều có cùng độ dài.

Theo hệ quả của Định nghĩa 2.13. Cây B trở thành cây tìm kiếm hoàn toàn cân bằng. Đặc biệt, tại $m = 4$, chúng ta nhận được 2-3-4-cây.



Hình 2.32. B-cây hàng 5.

Chúng ta sẽ xem xét một số thuộc tính và ứng dụng quan trọng hơn của B-cây. Đối với độc giả tò mò, có rất nhiều nguồn với thông tin bổ sung về các thuật toán và việc thực hiện các hoạt động cơ bản trên B-wood [Wirt-1980] [Shishkov-1995] [Knuth-3/1968].

B-cây được sử dụng rộng rãi trong thực tế, đặc biệt là trong các hệ thống quản lý thông tin khối lượng lớn (ví dụ: cơ sở dữ liệu quan hệ). Được biết, việc truy cập bộ nhớ ngoài trong hệ thống máy tính thường được thực hiện theo khối. Giả sử rằng trên một số mô hình đĩa cứng, một khu vực vật lý chứa chính xác 4096 byte. Sau đó, việc đọc và viết sẽ diễn ra trong các khối có kích thước này. Tuy nhiên, truy cập vào các đơn vị đĩa cứng chậm hơn đáng kể so với truy cập vào RAM, do đó, để tối ưu hóa hệ thống, cần giảm thiểu số lượng các cung được chuyển giữa RAM và đĩa cứng. Giả sử chúng ta phải tổ chức một lượng lớn thông tin không vừa trong RAM, và cần phải tìm kiếm một phần tử của nó bằng một khóa cho trước. Chức năng tìm kiếm thực hiện những việc sau: nó đọc một khối và tìm khóa nó cần trong đó. Nếu anh ta tìm thấy nó - anh ta hoàn thành công việc, nếu không anh ta đọc khối khác và tìm kiếm lại. Quá trình kết thúc nếu mục được tìm thấy hoặc không tồn tại. Do đó, các khối được đọc phải mang thông tin đến mức cần đọc một số khối bổ sung tối thiểu để xác định xem liệu khóa được tìm kiếm có tồn tại hay không.

Ứng dụng hiệu quả của B-cây trong các hệ thống như vậy bắt nguồn từ đặc điểm quan trọng sau: mỗi đỉnh k của B-cây chứa từ $[m/2] - 1$ đến $m - 1$ phần tử. Với một giá trị được chọn đúng của m , hóa ra một đỉnh chiếm đúng một khối bộ nhớ (trong trường hợp của chúng ta là một khu vực vật lý). Bởi vì B-cây cân bằng hoàn toàn, chiều cao của chúng tối đa là $h = \log_{[m/2]}(n + 1)/2$, mà ở các giá trị lớn hơn của m là một con số đủ nhỏ, thậm chí là rất lớn n . Các thuật toán tìm kiếm, thêm và xóa một phần tử có độ phức tạp là bậc $\Theta(h)$ và thậm chí nhiều hơn h chúng truy cập không quá h với số đỉnh từ B-cây.

Cần lưu ý rằng truy cập bộ nhớ chặn là điển hình không chỉ đối với các thiết bị lưu trữ bên ngoài. Đây cũng là trường hợp của hệ thống bộ nhớ ảo. Hệ quản trị cơ sở dữ liệu quan hệ Oracle sử dụng một sửa đổi của B-cây cổ điển để tổ chức và lưu trữ các chỉ mục của nó. Hệ điều hành Windows của Microsoft cũng sử dụng B-cây đã sửa đổi, bao gồm cả việc tổ chức hệ thống tệp.

Bài tập

- ▷ 2.24. Chứng minh rằng B-cây là cây tìm kiếm cân đối hoàn toàn.
- ▷ 2.25. So sánh B-cây, cây đẻ đen và cây Fibonacci.
- ▷ 2.26. Để xem xét các hoạt động cơ bản để làm việc với B-cây có thể được thực hiện như thế nào trong mô tả đã cho về cấu trúc của nó.

2.7. Bảng băm (H-bảng)

Bảng băm là một cấu trúc dữ liệu được đặc trưng bởi quyền truy cập trực tiếp vào các phần tử, bất kể kiểu của chúng. Độ phức tạp của các thao tác chính cơ bản (tìm kiếm, chèn, xóa và cập nhật) nói chung là không đổi, điều này làm cho nó cực kỳ hữu ích trong nhiều trường hợp. Khi định nghĩa bảng băm là một cấu trúc trừu tượng, chúng ta sẽ tự giới hạn trong ba phép toán cổ điển sau:

- **void** put(key, data) - đưa vào một phần tử
- data get(key); - tìm kiếm phần tử
- **void** remove(key) - Loại đi một phần tử

Như với danh sách duyệt và cây, bây giờ mỗi phần tử của bảng băm được đặc trưng bởi hai trường: khóa key và dữ liệu data. Khóa là

một định danh duy nhất: nếu hai phần tử có cùng một khóa, chúng được coi là giống hệt nhau.

Là một trường hợp đặc biệt của bảng băm, chúng ta có thể coi một mảng n phần tử trong đó chỉ số i ($i = 0, 1, \dots, n - 1$) là khóa của phần tử thứ i . Có thể thấy rằng hai phần tử có cùng khóa (chỉ mục) sẽ rơi vào cùng một phần tử của mảng. Ngoài ra, các thao tác đưa, xóa và tìm kiếm có độ phức tạp không đổi (do truy cập trực tiếp vào các phần tử của mảng). Trong thực tế, một mảng được triển khai khi thực hiện một bảng băm (tức là một cấu trúc có quyền truy cập trực tiếp đến các phần tử).

Quá trình mà khóa hằng số của một phần tử xác định địa chỉ của nó với độ phức tạp không đổi được gọi là băm.

2.7.1. Hàm băm (H-hàm số)

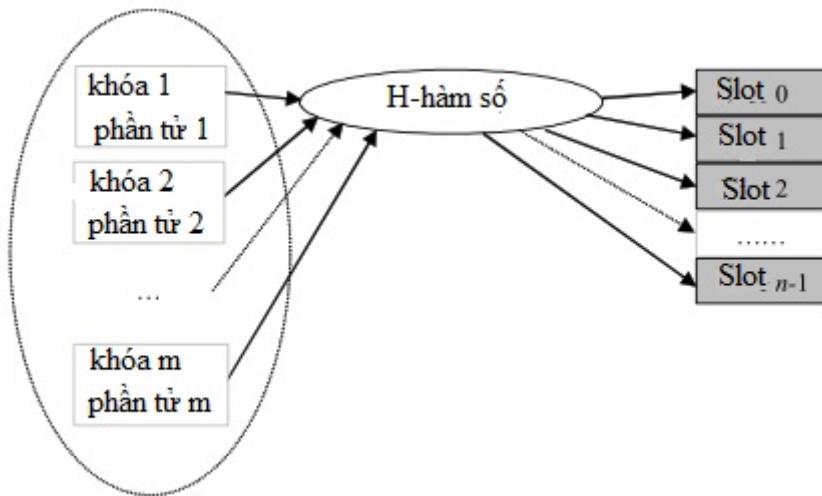
Chúng ta sẽ nhấn mạnh một lần nữa rằng khóa của phần tử có thể là bất kỳ cấu trúc nào xác định duy nhất nó. Ví dụ: nhân viên trong một công ty có thể tự nhận dạng mã PIN của họ (số tự nhiên gồm 10 chữ số, có thể có số 0 ở đầu). Trên thực tế, một nghiên cứu gần đây đã chỉ ra một số lỗi trong việc cấp mã PIN, bao gồm cả việc trùng lặp, khiến chúng trở thành khóa không đáng tin cậy.) Hoặc tên (chuỗi ký hiệu, nhưng chỉ với điều kiện bổ sung là không có hai người trùng tên). Chúng ta sẽ trình bày chi tiết hơn về ví dụ đầu tiên: chúng ta giả định rằng khóa là một số có 10 chữ số. Chúng ta sẽ sử dụng một mảng $a[]$ với n phần tử (số n được gọi là dung lượng bảng băm). Rõ ràng là nếu số lượng nhân viên nhỏ hơn hoặc bằng n , chúng ta sẽ có đủ bộ nhớ để ghi chúng vào mảng $a[]$. Tuy nhiên, để đạt được độ phức tạp không đổi của việc đưa vào, chúng ta phải có một hàm (hàm băm), hàm này trên khóa của mỗi phần tử để khớp duy nhất địa chỉ từ 0 đến $n - 1$. Nếu chúng ta coi các trường của một mảng là các vị trí (xem Hình 1), được đánh số từ 0 đến $n - 1$, trong đó một phần tử duy nhất có thể được đặt, thì cần có một hàm để xác định vị trí tương ứng bằng một khóa phần tử. .

Định nghĩa 2.14. Chúng ta xem xét một bảng băm với kích thước n và đặt U các giá trị cho phép đối với các khóa (vũ trụ). Sau đó, bảng

hàm băm, chúng ta sẽ hiểu hình ảnh:

$$h : U \rightarrow \{0, 1, \dots, n - 1\}$$

Ví dụ, đối với bất kỳ số k (PIN) có 10 chữ số nào, có thể lấy một số nguyên từ 0 đến $n - 1$ bằng phép toán $k \% n$. Phần còn lại thu được bằng cách chia k cho dung lượng của bảng băm xác định một hàm băm khả thi. Trong trường hợp này, đặc điểm quan trọng thứ hai của bảng băm được chú ý: chúng ta có thể thấy mình ở trong tình huống mà các phần tử có các khóa khác nhau được ánh xạ tới cùng một địa chỉ từ mảng. Ví dụ, với $k_1 = 8004104369, k_2 = 8004102469$ và $n = 100$ ta có: $k_1 \% n = k_2 \% n = 69$, tức là các phần tử có hai khóa khác nhau nhận cùng một địa chỉ băm. Sự xuất hiện của một tình huống như vậy được gọi là xung đột và là một vấn đề lớn đối với bảng băm.



Hình 2.33. Hàm băm khớp với địa chỉ của từng khóa trong bảng băm.

2.7.2. Sự xung đột

Định nghĩa 2.15. Nếu cho hàm băm h và hai khóa $k_1, k_2 \in U$ thỏa mãn $h(k_1) = h(k_2)$, thì ta nói rằng các phần tử k_1 và k_2 đang xung đột. Chúng ta cũng sẽ nói rằng chúng đồng nghĩa.

Không khó để thấy rằng hàm băm xác định một *quan hệ tương đương*: tập hợp các từ đồng nghĩa của cùng một lớp và chỉ chúng tạo thành một lớp *tương đương*.

Nguyên lý (nguyên lý Dirichlet, nguyên tắc ngắn kéo): n ngắn kéo được đưa ra. Nếu chúng ta đặt $n + 1$ đối tượng vào chúng, thì sẽ có ít nhất hai đối tượng trong ít nhất một ngắn kéo.

Xung đột là không thể tránh khỏi - ngay cả khi chúng ta tạo một hàm băm "hoàn hảo", xung đột được đảm bảo sẽ xảy ra muộn nhất khi số lượng phần tử được đưa vào vượt quá khả năng của bảng băm (theo nguyên tắc Dirichlet). Trong thực tế, vụ xung đột sẽ xảy ra sớm hơn nhiều (như thể theo Định luật Murphy, nhưng thực tế là theo thống kê). Một ví dụ minh họa cho trường hợp thứ hai là "ví dụ về ngày tháng năm sinh": Xác suất để trong số 23 người có hai người sinh cùng ngày (tức là cùng ngày và tháng sinh) là bao nhiêu? Hóa ra xác suất này là một số trên $\frac{2}{1}$ một chút: nó được tính là

$$1 - \prod_{i=1}^{23} \frac{366 - i}{365} = 0,5063.$$

Phương trình trên thu được như thế nào?

Hãy kí hiệu A là biến cố "*không có hai người trùng ngày sinh*". Khi đó xác suất yêu cầu sẽ bằng $1 - P(A)$. Để tính $P(A)$, chúng ta sẽ sử dụng định nghĩa cổ điển của xác suất: tỷ số giữa số sự kiện thuận lợi trên tổng số sự kiện cơ bản. Trong trường hợp của chúng ta, thuận lợi (liên quan đến A) là các sự kiện không có hai người (trong số 23 người) có cùng ngày sinh và họ có số lượng là 365, 364, 363...343. Đồng thời, tất cả các loại cấu hình ngày tháng năm sinh đều được đưa ra bởi số 365²³. Do đó công thức trên dễ dàng làm theo. Như vậy, cơ hội chiến thắng sẽ nghiêng về phía chúng ta nếu đặt cược vào kèo cuối cùng trong bữa tiệc có đủ số lượng khách mời. Ở đây chúng ta đã giả định rằng một năm có 365 ngày, tức là nó không cao.

Như vậy, ngoài mong muốn chọn một hàm băm tốt, tức là dẫn đến một số xung đột tối thiểu, vẫn đề giải quyết chúng là điều tối quan trọng. Trong hai điểm tiếp theo, chúng ta sẽ xem xét các hàm băm được sử dụng phổ biến nhất, cũng như các cách tiếp cận phổ biến nhất để đối phó với xung đột. Trong 2.5.3. chúng ta cũng sẽ đưa ra hai ví dụ triển khai bảng băm.

2.7.3. Hàm băm cổ điển

Việc lựa chọn một hàm băm có phù hợp hay không được xác định bởi hai điều: hàm băm không mất nhiều thời gian tính toán và phân phối các phần tử tương đối đồng đều trong các địa chỉ băm. Chúng ta sẽ giả định rằng khóa của các phần tử là số nguyên. Nếu không, chúng ta có thể so sánh chúng theo một số quy tắc: ví dụ: nếu các khóa là chuỗi, chúng ta có thể tính tổng các mã ASCII tương ứng. Do đó, đối với chuỗi "hello", chúng ta sẽ nhận được $104 + 101 + 108 + 108 + 111 = 532$. Nếu chúng ta chắc chắn rằng chuỗi chỉ bao gồm các chữ cái Latinh viết thường, thì chúng ta có thể trừ từ mã ASCII của mỗi chữ cái ASCII. mã của chữ cái 'a'. Điều này sẽ tránh bị tràn nếu chuỗi quá dài. Trong một số trường hợp, cách chuyển đổi từ khóa chuỗi thành số nguyên được mô tả dẫn đến kết quả rất kém. Ví dụ, đối với hàm băm ở trên, bất kỳ hoán vị nào của các ký tự sẽ cho cùng một mã băm; các chuỗi ngắn sẽ ở đầu bảng băm. Do đó, đôi khi số kết quả phụ thuộc vào vị trí của ký tự trong chuỗi là thích hợp. Nếu không có ràng buộc cộng đồng, chúng ta có thể giả định rằng chúng ta đang làm việc với các khóa số nguyên: mỗi chuỗi ký tự $a_1a_2\dots a_n$ có thể được coi là một số được viết trong hệ thống số gồm b chữ số (trong đó b là số các ký tự hợp lệ khác nhau).

Cho một bảng băm với dung lượng n và phần tử có khóa k . Dưới đây chúng ta sẽ liệt kê các hàm băm phổ biến nhất trong thực tế (trên một khóa số nguyên).

Lượng dư khi chia theo kích thước của bàn.

Ở trên chúng ta đã đề cập đến cách băm đơn giản và đồng thời khá hiệu quả này - khóa k được chia nguyên cho n và lấy phần dư của phép chia. Trong cách tiếp cận này, không phù hợp với dung lượng của bảng băm khi chọn số n , bậc của cặp: nếu $n = 2^p$, thì mã băm của một khóa k sẽ là p bit ít nhất của k . Ví dụ, cho $n = 2^4 = 16$ và $k = 173$. Biểu diễn nhị phân của 173 là 10101101. Phần còn lại $173\%16$ là 13, phần cuối cùng được viết trong hệ thống số nhị phân là 1101 - tức là chính xác là 4 bit thấp hơn của 173. (xem 1.1.6.)

Phép băm này sẽ hoạt động không tốt nếu phân phối khóa có một số lượng lớn các số với các bit thấp hơn phù hợp. Nói chung,

điều tốt là mã băm kết quả phụ thuộc vào tất cả các bit của khóa. Với mục đích này, một số nguyên tố thường được chọn cho dung lượng bảng băm.

Phép băm nhân đôi

Một phương pháp phổ biến khác là *băm nhân*. Một hằng số thực a , $0 < a < 1$. Đổi với một khóa k cho trước, hàm băm có dạng:

$$h(k) = \lfloor n \cdot \{k \cdot a\} \rfloor$$

Ở đây chúng ta biểu thị bằng $\{k \cdot a\}$ phần phân số của số thực, tức là $k \cdot a - \lfloor k \cdot a \rfloor$.

Mặc dù lựa chọn hằng số a (với ràng buộc $0 < a < 1$) là tùy ý, đổi với một số giá trị, kết quả thực tế tốt hơn. Knuth [Knuth-3/1968] đề xuất sử dụng tỷ lệ vàng (xem 1.2.2.):

$$a = \frac{\sqrt{5} - 1}{2} = 0,6180339887\dots$$

Hàm băm trên các bộ phận khóa

Trích số

Trong lược đồ này, chỉ các chữ số nằm ở các vị trí nhất định mới được trích xuất từ khóa (ví dụ, chúng ta có thể lấy các chữ số đầu tiên, thứ ba và thứ năm của số). Như vậy, với các khóa 123569, 425435, 546754, 676576 ta sẽ nhận được địa chỉ băm lần lượt là 136, 453, 565, 667. Có thể thấy đối với trường hợp cụ thể n phải là 1000 (vì khi giải nén chữ số 1, 3 và 5 chúng ta có thể nhận được một số trong phạm vi $[0, 999]$). Phương pháp này hoạt động tốt khi các số không chứa nhiều chữ số lặp lại.

Gấp

Phương pháp này thường được sử dụng nhất khi các khóa có số lượng rất lớn. Có thể có sự đa dạng, nhưng nhìn chung tất cả đều dựa trên việc chia khóa thành các phần và thực hiện một số phép toán số học trên các phần kết quả. Ví dụ, số có thể được chia thành hai (hoặc ba hoặc nhiều phần) và tổng các số thu được có thể xác định địa chỉ băm.

Nâng giữa lên thành hình vuông

Lược đồ này dựa trên việc trích xuất các chữ số p ở giữa của khóa và bình phương chúng. Ví dụ: đối với khóa 125657134280980, ba chữ số trung bình là 134. Chúng ta bình phương chúng: $134 \cdot 134 = 17956$. Nếu kết quả vượt quá n, một vài chữ số có nghĩa đầu tiên bị loại bỏ: ví dụ: nếu $n = 10001$, thì từ 17956 chữ số đầu tiên bị loại bỏ và địa chỉ băm kết quả là 7956. Lưu ý rằng phép toán cuối cùng không tương đương với việc tìm phần dư khi chia cho n .

So sánh một số hàm băm được xem xét

Để so sánh các phương pháp đã trình bày, chúng ta sẽ áp dụng kết quả của một thử nghiệm thực tế: Chúng ta xem xét một bảng băm với kích thước n và m số PIN ngẫu nhiên, chúng ta sẽ sử dụng làm khóa (được tạo theo cách chúng hợp lệ và ngày sinh là từ thế kỷ trước). Mục tiêu của chúng ta sẽ là so sánh sự phân bố của mã băm khi sử dụng từng hàm băm được thảo luận ở trên. Kết quả của thí nghiệm (tại $m = 1031000, n = 1031$) được trình bày trong Bảng ??, Trong đó $\chi^2 = \frac{n}{m} \sum_{i=1}^n \left(f_i - \frac{m}{n} \right)^2$, và f_i là số khóa có mã băm, bằng i .

Hàm băm	χ^2
Phần dư khi chia cho 1031	729
Trích xuất các chữ số (số được tạo bởi 3 chữ số cuối cùng của mã PIN)	352
Gấp (chia mã PIN thành 1-3-3-3 và tổng các số thu được)	735
Bình phương số được tạo bởi ba chữ số trung bình của mã PIN.	233

Hình 2.34. χ^2 so sánh các hàm băm trên PIN, $n = 1031, m = 1031000$.

Đối với thống kê χ^2 , người ta biết rằng nếu hàm băm là ngẫu nhiên (tức là nó được mong đợi hoạt động "tốt" như nhau trên bất kỳ bộ khóa nào, không chỉ đối với các số là mã PIN hợp lệ) và $m > cn$, thì χ^2 phải bằng $n \pm \sqrt{n}$ với xác suất $1 - 1/c$.

Có thể thấy rằng đối với ví dụ đã chọn (khóa PIN) và các giá trị $n = 1031$ và $m = 1031000$, phương pháp thứ ba và thứ nhất là hiệu quả nhất, phương pháp thứ hai và thứ tư dẫn đến băm không đồng

đều hơn.

Hàm băm trên chuỗi

Chuỗi ký tự là kiểu dữ liệu được băm phổ biến nhất. Đồng thời, việc tìm kiếm một hàm băm tốt là một vấn đề nghiêm trọng. Dưới đây là một số hàm băm được sử dụng phổ biến nhất cho các chuỗi ký tự, không có nghĩa là đã đầy đủ và không làm phiền người đọc về các chi tiết không cần thiết. Hoạt động của các hàm sẽ được giải thích tổng hợp trên cơ sở các đoạn chương trình cụ thể. Trước đó, chúng ta sẽ đưa ra sơ đồ chung nhất:

```
result = khởi tạo ();
while (c = next_character ()) {
    result = kết hợp (result, c);
    result = internal_modification (result);
}
result = add_modification (result);
```

Băm phụ gia

Đây là cách đơn giản nhất, phổ biến nhất (cổ điển), nhưng tiếc là cách băm kém hiệu quả nhất. Các mã ASCII của các ký hiệu được tính tổng và tổng được coi là một mô-đun của kích thước mảng.

```
unsigned long hashFunction(const char *key, unsigned long size)
{ unsigned long result = 0;
while (*key)
    result += (unsigned char) *key++;
return result % size;
}
```

Thông thường độ dài chuỗi được bao gồm trong số lượng để nó có thể ảnh hưởng rõ ràng đến mã băm:

```
unsigned long hashFunction(const char *key, unsigned long size)
{ unsigned long result = strlen (key);
while (*key)
    result += (unsigned char) *key++;
return result % size;
}
```

Lưu ý rằng độ dài của chuỗi ký tự có thể được truyền dưới dạng tham số, điều này giúp tiết kiệm quyền truy cập đắt tiền vào hàm

`strlen()`. Hoặc chỉ đơn giản là có được sự khác biệt giữa vị trí hiện tại trước và sau khi duyệt chuỗi (tuy nhiên, điều này không phải lúc nào cũng khả thi với các hàm băm tiếp theo, vì chúng thực hiện các thao tác result phức tạp hơn trong phần thân của vòng lặp):

```
unsigned long hashFunction(const char *key, unsigned long size)
{ const char *saveKey = key;
  unsigned long result = 0;
  while (*key)
    result += (unsigned char) *key++;
  result += saveKey - key;
  return result % size;
}
```

Kích thước của mảng thường được chọn là một số nguyên tố. Tuy nhiên, đôi khi một mảng có kích thước 2 được sử dụng, trong đó hàng cuối cùng có thể được đơn giản hóa thành:

`return result & (size - 1);`

hoặc tốt hơn:

`hash = (hash ^ (hash>>10) ^ (hash>>20)) & mask;`

Kỹ thuật này cũng hợp lệ cho các hàm bên dưới, có chứa dòng cuối cùng như vậy.

Băm xoay vòng

Không có bộ sưu tập ở đây, chỉ có hoạt động hàng loạt. Kích thước của mảng phải là một số nguyên tố.

```
unsigned long hashFunction(const char *key, unsigned long size)
{ unsigned long result = strlen(key);
  while (*key)
    result = (result << 4) ^ (result >> 8) ^ ((unsigned char) *key++);
  return result % size;
}
```

Băm từng cái một

Hàm là một biến thể của hàm trên nhưng với nhiều thao tác hơn trên kết quả biến, được thực hiện riêng biệt.

```
unsigned long hashFunction(const char *key, unsigned long size)
```

```

{ unsigned long result = 0;
  while (*key) {
    result += (unsigned char) *key++;
    result += result << 10;
    result ^= result >> 6;
  }
  result += result << 3;
  result ^= result >> 11;
  result += result << 15;
  return result % size;
}

```

Băm Pearson

Một mảng bổ sung tab[] được sử dụng ở đây, chứa hoán vị các số từ 0 đến 255. Lưu ý rằng mã băm kết quả là một byte: 0 đến 255. Có thể nhận được mã lớn hơn nếu hàm được gọi nhiều lần với các mảng khác nhau tab[], trong đó mỗi lệnh gọi sẽ cho một byte kết quả.

```

unsigned char hashFunction(const char *key, unsigned long size,
  const unsigned char tab[])
{
  unsigned long result = strlen(key);
  while (*key)
    result = tab[result ^ ((unsigned char) *key++)];
  return result;
}

```

Băm CRC

Điều này yêu cầu các giá trị trong mảng được tạo bởi *Linear Feedback Shift Register*. Không đi vào quá nhiều chi tiết, chúng ta sẽ đề cập rằng cái sau là một thiết bị tạo ra một chuỗi nhị phân giả ngẫu nhiên thỏa mãn một số điều kiện nhất định.

```

unsigned long hashFunction(const char *key, unsigned long size,
  const unsigned long tab[])
{
  unsigned long result = strlen(key);
  while (*key)
    result = (result << 8) ^ tab[(result >> 24) ^ ((unsigned char) *key
      ++)];
  return result % size;
}

```

}

Tổng quát CRC băm

Tương tự như trên, nhưng tab[] có thể chứa các giá trị tùy ý.

Băm phổ quát

Tuy nhiên, dù chúng ta chọn hàm băm nào thì trong trường hợp xấu nhất, nó sẽ gây ra $\Theta(n)$ xung đột trong đó n là kích thước của bảng băm. Tất nhiên, điều này sẽ được bù đắp bằng các cuộc gọi lặp đi lặp lại đến hàm, vì trường hợp xấu sẽ hiếm khi xảy ra.

Tuy nhiên, chúng ta có thể làm gì để bảo vệ mình khỏi những xung đột thường xuyên hơn? Câu trả lời là: để đảm bảo sự phân bố đồng đều của tập các giá trị cho phép của các khóa trên tập các chỉ số của mảng. Sau đó, xác suất so sánh cùng một chỉ mục trong một mảng cho hai chuỗi khác nhau, tức là một xung đột, sẽ là $1/n$.

Nhưng làm thế nào để đạt được nó? Một khả năng là kết hợp một số hàm băm để phân phối đồng đều các phần tử. Được rồi, nhưng có bao nhiêu tính năng như vậy để chọn? Gọi số giá trị cho phép khác nhau của khóa là m ($m = |U|$), và số hàm ta cần - f . Khi đó chúng ta sẽ muốn $f/m = 1/n$. nghĩa là, chúng ta sẽ cần m/n của một số hàm khác nhau, mỗi hàm phân phối đều các khóa trên các chỉ số của mảng.

Định nghĩa 2.16. Tập hợp $H = \cup_a \{h_a\}$ của các hàm băm được gọi là *tập hợp phổ quát của các hàm băm* nếu:

- $|H| = m/n$ - chứa đúng m/n trong hàm số h_a
- $P(h_a(x) = h_a(y)) = 1/n$ - xác suất xung đột là $1/n$

Một cách khả thi để có được một tập hợp phổ quát các hàm băm là chọn m/n hàm được định nghĩa là (a là một tham số khác nhau cho mỗi hàm):

$$h_a(k) = \left(\sum_{i=0}^r a_i k_i \right) \bmod n$$

ở đâu:

$k = (k_0, k_1, \dots, k_r)$ đang phá khóa k thành $r + 1$ phần (ví dụ: byte)
 $a = (a_0, a_1, \dots, a_r)$, một vectơ có các thành phần được chọn ngẫu nhiên từ $0, 1, \dots, n - 1$.

Sau đây là một ví dụ thực hiện. Mảng tab[] chứa bao nhiêu phần tử bằng số bit tối đa trong chuỗi đầu vào, được chọn ngẫu nhiên từ $\{0, 1, \dots, n - 1\}$. Chúng ta rời các chi tiết để người đọc.

```
unsigned long hashFunction(const char *key, unsigned long size,
    const unsigned long tab[MAXBITS])
{
    unsigned char k;
    unsigned i;
    unsigned long result;
    unsigned long l3 = (result = strlen(key)) << 3;
    for (i = 0; i < l3; i += 8)
    {
        k = (unsigned char) key[i >> 3];
        if (k&0x01) result ^= tab[i+0];
        if (k&0x02) result ^= tab[i+1];
        if (k&0x04) result ^= tab[i+2];
        if (k&0x08) result ^= tab[i+3];
        if (k&0x10) result ^= tab[i+4];
        if (k&0x20) result ^= tab[i+5];
        if (k&0x40) result ^= tab[i+6];
        if (k&0x80) result ^= tab[i+7];
    }
    return result % size;
}
```

Hàm băm Zobrist

Ở đây, bảng tab[][] là hai chiều và các giá trị lại được chọn ngẫu nhiên từ $\{0, 1, \dots, n - 1\}$. Lựa chọn cẩn thận có thể dẫn đến băm phổ quát.

```
unsigned long hashFunction(const char *key, unsigned long size
    ,
    const unsigned long tab[MAXBYTES][256])
{
    unsigned i;
    unsigned long result = strlen(key);
    for (i = 0; i < len; i++)
        result ^= tab[i][*key++];
    return result % size;
}
```

Bài tập

- ▷ 2.27. 1. So sánh các hàm băm được thảo luận ở trên với:
- độ phức tạp tính toán
 - tính đồng nhất của phân phối chính
- ▷ 2.28. Những vấn đề nào mà hàm băm gấp phải trên các bộ phận quan trọng?
- ▷ 2.29. Bạn thấy ưu và nhược điểm nào đối với mỗi hàm băm được xem xét cho chuỗi ký tự?
- ▷ 2.30. Có phải một hàm băm phân phối các phần tử đồng đều hơn luôn luôn tốt hơn?
- ▷ 2.31. Để so sánh hàm băm phổ quát và hàm băm của Zobrist.
- ▷ 2.32. Tập hợp m/n được cho bởi số hàm băm được xác định là (a là một tham số khác nhau đối với mỗi hàm):

$$h_a(k) = \left(\sum_{i=1}^r a_i k_i \right) \bmod n,$$

ở đâu:

$k = (k_0, k_1, \dots, k_r)$ đang chia khóa x thành $r + 1$ phần (ví dụ: byte)

$a = (a_0, a_1, \dots, a_r)$, một vectơ có các thành phần được chọn ngẫu nhiên từ $\{0, 1, \dots, n - 1\}$.

n - kích thước của bảng băm

m - lũy thừa của tập các giá trị cho phép của khóa.

Dựa trên Định nghĩa ??, Để chứng minh rằng đây là một tập hợp phổ quát của các hàm băm.

- ▷ 2.33. Bổ sung Bảng ?? với băm nhân.

- ▷ 2.34. Để xem xét có thể áp dụng sự kết hợp (hoặc sửa đổi) nào của các phương pháp được xem xét để thu được kết quả tốt hơn nữa (trên một bộ mã PIN được chọn ngẫu nhiên, n và m)? χ^2 có thể được mong đợi là $n \pm \sqrt{n}$ nếu một hàm băm được biên dịch hoạt động tối ưu cho hàm băm mã PIN cụ thể không?

- 2.35. Lặp lại thí nghiệm đã mô tả cho các giá trị khác nhau của n và m . Các kết quả trên đã được xác nhận chưa? Khi tạo dữ liệu thử nghiệm, có thể sử dụng thuật toán để kiểm tra tính đúng đắn của số PIN từ Bài toán 1.203.
- 2.36. Bạn có đồng ý với phương pháp lấy dữ liệu cho bài kiểm tra diễn dữ liệu từ Hình ?? (tạo với một hàm cho các số giả ngẫu nhiên, nhưng với phân phối nào ...)? Điều này có ảnh hưởng gì đến kết quả cuối cùng? Bạn có thể đề xuất một giải pháp cho vấn đề?

2.7.4. Đối phó với xung đột

Chúng ta sẽ minh họa các sơ đồ quản lý xung đột khác nhau bằng một ví dụ cụ thể. Hãy đưa ra một bảng băm có dung lượng $n = 10$. Chúng ta sẽ sử dụng một hàm băm đơn giản dựa trên việc trích xuất các chữ số - chữ số đầu tiên của khóa sẽ là địa chỉ băm của chúng ta. Trong ví dụ, các mục chúng ta muốn đưa vào là các khóa 234, 235, 567, 123, 534 và 647.

Hàm băm đã đóng

Kiểm tra tuyến tính

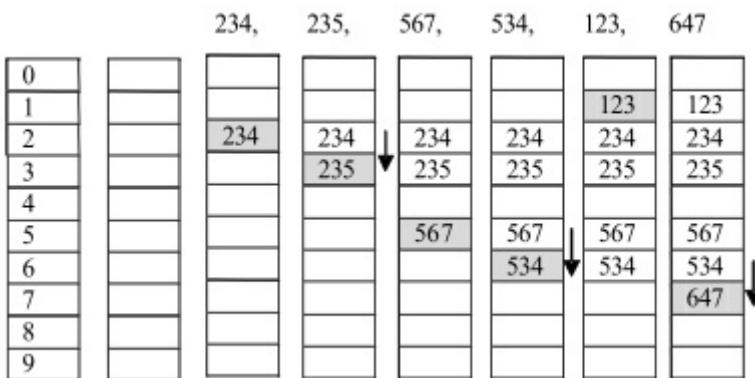
Kiểm tra tuyến tính, kiểm tra bậc hai và băm kép là một phần của sơ đồ giải quyết xung đột tổng quát hơn được gọi là *băm đóng*. Với nó, chúng ta có một nơi chính để lưu trữ dữ liệu (n "slot", trái ngược với việc triển khai có một phần bổ sung cho các xung đột). Khi xung đột xảy ra, chúng ta cố gắng thay đổi địa chỉ băm kết quả một cách tuần tự cho đến khi chúng ta đạt được vị trí trống. Sự thay đổi được thực hiện theo một sơ đồ xác định trước.

Ví dụ, có thể tăng địa chỉ của một số tự nhiên s ($0 < s < n$). Nếu địa chỉ trở nên bằng n , chúng ta tiếp tục tìm kiếm ở đầu bảng băm, lấy phần dư modulo n . Để có thể thu thập thông tin tất cả các địa chỉ của bảng băm ở độ phóng đại như vậy, n và s phải nguyên tố lân nhau, tức là $GNP(n, s) = 1$.

Rõ ràng, cách tiếp cận này không thể bao gồm nhiều phần tử hơn dung lượng của bảng băm (xem Hình 2.35) và khi nó đầy, cần phải mở rộng nó (cấp phát bộ nhớ cho một mảng lớn hơn và giải phóng phần bị chiếm từ mảng bộ nhớ hiện tại). Chúng ta sẽ xem điều này được triển khai như thế nào trong thực tế trong lần triển khai thứ

hai của bảng băm bên dưới (xem 2.5.3).

Chèn phần tử:



Hình 2.35. Bao gồm tuần tự của một phần tử. Cho phép xung đột với thử nghiệm tuyển tính với bước 1.

Khi tìm kiếm một phần tử theo khóa, hãy tiến hành theo cách tương tự - đầu tiên địa chỉ băm được tính bằng hàm băm và trong trường hợp không tìm thấy khóa được yêu cầu, kiểm tra tuyển tính (thay đổi địa chỉ) được thực hiện cho đến khi đạt được yêu cầu phần tử. Sau đó giả định rằng khóa nằm trong bảng băm.

Trường hợp $s = 1$ là phổ biến: trong trường hợp va chạm, người ta di chuyển đến địa chỉ tiếp theo, v.v. với bước 1. Trong trường hợp này, cũng như trong trường hợp tổng quát hơn, có thể gặp các khóa mà việc băm đã tỏ ra không hiệu quả. Ví dụ: khi các phần tử tạo thành nhóm với mã băm gần nhau (cái gọi là "cụm"). Một loạt các va chạm mới (thứ cấp) sau đó phát sinh trong quá trình giải quyết va chạm (xem Hình 2.35). Dưới đây chúng ta sẽ xem xét hai kiểu băm đóng quan trọng khác.

Kiểm tra bậc hai

Trong *thử nghiệm bậc hai*, như tên cho thấy, một bước của loại $c_1i + c_2i^2, c_2 \neq 0$. Độ lệch phụ thuộc bậc hai vào số sê-ri của mẫu i . Phương pháp này hoạt động hiệu quả hơn đáng kể so với kiểm tra tuyển tính, mặc dù nó cũng cho thấy những tác động tiêu cực trong trường hợp một cụm khóa (tích lũy thứ cấp của các khóa). Tuy nhiên, vấn đề chính của nó là nó *không* đảm bảo thu thập dữ

liệu toàn bộ bảng, tức là anh ta có thể không tìm thấy chỗ trống, mặc dù có một chỗ. [Cormen, Leiserson, Rivest-1997].

Băm kép

Hàm băm kép tương tự như hai phương pháp được xem xét, sử dụng hai hàm băm:

$$h(k, i) = h_1(k) + i.h_2(k)$$

Đây i là số mẫu sau lần xung đột đầu tiên. Hàm băm thứ hai chỉ được sử dụng trong trường hợp kết quả của lần đầu tiên dẫn đến xung đột. Cả khóa gốc của phần tử và địa chỉ băm thu được khi áp dụng hàm băm đầu tiên đều có thể được sử dụng làm khóa để băm lại.

Mở băm

Bộ nhớ bổ sung cho các xung đột

Như tên cho thấy, lược đồ này cung cấp thêm bộ nhớ để xử lý xung đột. Mỗi phần tử xung đột nằm trong không gian trống đầu tiên trong bộ nhớ phụ (xem Hình 2.36).

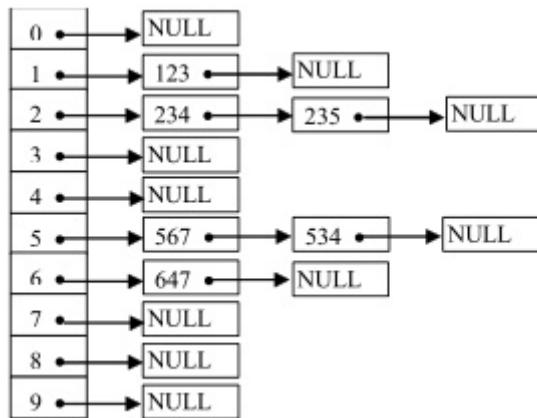
Chèn phần tử:

	234,	235,	567,	534,	123,	647
0						
1						
2	234	234				
3						
4						
5			567	567		
6						
7						
8						
9						
10			235	235	235	235
11				534	534	534
...

Hình 2.36. Giải quyết xung đột bằng cách phân bổ bộ nhớ bổ sung.

Khi tìm kiếm trong bảng băm, địa chỉ thu được từ hàm băm sẽ được kiểm tra đầu tiên và nếu phần tử có khóa được tìm kiếm không có ở đó, thì toàn bộ phần bổ sung sẽ được tìm kiếm.

Danh sách tràn



Hình 2.37. Kết nối động: bảng băm, sau khi bao gồm các phần tử 234, 235, 567, 123, 534, 647.

Cũng có thể sử dụng bộ nhớ được cấp phát động bên ngoài: Các phần tử có địa chỉ băm phù hợp nằm trong một số cấu trúc dữ liệu chuẩn. Cấu trúc được sử dụng phổ biến nhất nhưng không nhất thiết là hiệu quả nhất cho mục đích này là danh sách liên kết. Cách tiếp cận này để xử lý các xung đột thường được gọi là danh sách tràn: mỗi vị trí trong bảng băm là một con trỏ đến danh sách liên kết động chỉ chứa các thành viên của lớp từ đồng nghĩa tương ứng. Khi một phần tử mới được đưa vào, nó sẽ được thêm vào đầu danh sách nằm trên vị trí được xác định bởi hàm băm (xem Hình 2.37). Do đó, việc cẩn thận khi va chạm đương nhiên bị loại bỏ.

Tương tự, khi tìm kiếm, danh sách được xác định bởi hàm băm sẽ được xem. Độ phức tạp của thao tác tìm kiếm sau đó phụ thuộc vào số lượng phần tử mà phần tử đã cho va chạm với nhau, tức là vào độ dài của danh sách liên kết động tương ứng.

Các phần tử va chạm có thể được lưu trữ trong một cấu trúc trong đó tìm kiếm ít mang tính thuật toán hơn so với cấu trúc của danh sách tuyến tính - ví dụ: cây tìm kiếm nhị phân. Do đó, mỗi trường trong bảng băm sẽ là một con trỏ tới gốc của một cây nhị phân riêng biệt.

Bài tập

- ▷ 2.37. Để so sánh băm mở và đóng: ưu điểm và nhược điểm. Khi nào mỗi người trong số họ nên được ưu tiên?
- ▷ 2.38. Tại sao kiểm tra bậc hai không thể đảm bảo truyền toàn bộ mảng? Điều này có đúng với tất cả các hàm bình phương không?

2.7.5. Triển khai bảng băm

Chúng ta sẽ xem xét hai cách triển khai khác nhau của một bảng băm. Đầu tiên, khóa của bảng băm là số nguyên (điều này quan trọng trong việc thực hiện hàm băm, sẽ theo nguyên tắc chia cho phần dư). Chúng ta sẽ giải quyết các vụ va chạm bằng cách sử dụng một danh sách được liên kết động. Điều này sẽ minh họa cấu trúc dữ liệu hiệu quả có thể được xây dựng nhanh chóng và dễ dàng như thế nào (lưu ý rằng loại trừ việc triển khai danh sách được liên kết, mã bổ sung cho bảng băm rất ngắn).

Dung lượng của bảng băm N được đặt ở đầu chương trình dưới dạng macro. Chi tiết triển khai có thể được tìm thấy dưới dạng nhận xét trong mã nguồn.

Chương trình 2.5. Bảng băm (206hash.c)

```
#include <stdio.h>
#include <stdlib.h>
#define N 211
typedef int data;
typedef long keyType;
#define NOT_EXIST (-1) /* trả về từ get () khi thiếu phần tử*/
struct list {
    keyType key;
    data info;
    struct list *next;
};
struct list *hashTable[N];

/* bao gồm một phần tử ở đầu danh sách được liên kết*/
void insertBegin(struct list **L, keyType key, data x)
{
    struct list *temp;
```

```

temp = (struct list *) malloc(sizeof(*temp));
if (NULL == temp) {
    fprintf(stderr, "Không đủ bộ nhớ cho mục mới!\n");
    return;
}
temp->next = *L;
(*L) = temp;
(*L)->key = key;
(*L)->info = x;
}

/* xóa một phần tử khỏi danh sách */
void deleteNode(struct list **L, keyType key)
{
    struct list *current = *L;
    struct list *save;
    if ((*L)->key == key) { /*phần tử đầu tiên phải được xóa*/
        current = (*L)->next;
        free(*L);
        (*L) = current;
        return;
    }

    /* tìm phần tử cần xóa */
    while (current->next != NULL && current->next->key != key)
        current = current->next;
    if (NULL == current->next) {
        fprintf(stderr, "Lỗi: Không tìm thấy phần tử cần xóa! \n");
        return;
    }
    else {
        save = current->next;
        current->next = current->next->next;
        free(save);
    }
}

/*tìm kiếm một phần tử trong danh sách được liên kết*/
struct list* search(struct list *L, keyType key)
{
    while (L != NULL) {
        if (L->key == key) return L;
}

```

```

        L = L->next;
    }
    return NULL;
}

unsigned hashFunction(keyType key)
{ return(key % N); }

void initHashTable(void)
{ unsigned i;
  for (i = 0; i < N; i++) hashTable[i] = NULL; }

void put(keyType key, data x)
{ int place = hashFunction(key);
  insertBegin(&hashTable[place], key, x); }

data get(keyType key)
{ int place = hashFunction(key);
  struct list *l = search(hashTable[place], key);
  return (NULL != l) ? l->info : NOT_EXIST; }

int main()
{
  initHashTable();
  put(1234, 100); /* -> trong slot 179 */
  put(1774, 120); /* -> trong slot 86 */
  put(86, 180); /* -> trong slot 86 -> xung đột */
  printf("In dữ liệu cho phần tử có khóa 86: %d \n", get(86));
  printf("In dữ liệu cho phần tử có khóa 1234: %d \n", get(1234));
  printf("In dữ liệu cho phần tử có khóa 1774: %d \n", get(1774));
  printf("In dữ liệu cho phần tử có khóa 1773: %d \n", get(1773));
  return 0;
}

```

Kết quả thực hiện chương trình:

In dữ liệu cho phần tử có khóa 86: 180
 In dữ liệu cho phần tử có khóa 1234: 100
 In dữ liệu cho phần tử có khóa 1774: 120
 In dữ liệu cho phần tử có khóa 1773: -1

Ví dụ thứ hai, chúng ta sẽ xem xét một bài toán thực tế cụ thể được giải quyết hiệu quả bằng cách sử dụng bảng băm. Trong phần

triển khai này, chúng ta sẽ tập trung vào một số vấn đề cụ thể hơn: băm chuỗi ký tự, mở rộng bảng băm khi mảng được cấp phát trước không đủ và hơn thế nữa.

Bài toán: Một văn bản (danh sách các từ) được đưa ra, cách nhau bởi một khoảng trắng. Tìm tần suất xuất hiện của mỗi từ trong văn bản.

Bài toán được mô tả thường phát sinh như một phần của các vấn đề thực tế phức tạp hơn. Ví dụ, phân tích một tài liệu từ một công cụ tìm kiếm (ví dụ: Google), trước khi xử lý thêm, hầu như nhất thiết phải bao gồm việc tìm số lần xuất hiện của mỗi từ trong tài liệu.

Để giải quyết vấn đề một cách hiệu quả với bảng băm, một số điều cơ bản cần được xem xét:

- Khóa bảng băm sẽ là một chuỗi ký tự - đây là một từ trong tài liệu. Ngoài ra, tần suất xuất hiện của từ sẽ được giữ nguyên. Đối với một số tác vụ, có thể không có dữ liệu bổ sung. Khi đó cấu trúc được gọi là **tập hợp băm**.
- Các xung đột sẽ được giải quyết bằng thử nghiệm tuyển tính, kích thước của bảng băm n sẽ là lũy thừa của 2. Lưu ý rằng trong triển khai để xuất, vị trí của biểu tượng là quan trọng, tức là abc và bca sẽ có các mã băm khác nhau. Khi xem xét một hàm băm dựa trên phần dư khi chia cho một số nguyên tố trong 2.5.1, chúng ta đã chỉ ra bằng một ví dụ cụ thể rằng đây không phải là một cách tiếp cận hiệu quả, vì mã băm không phụ thuộc vào tất cả các bit của khóa. Tuy nhiên, trong trường hợp này, đây không phải là vấn đề nghiêm trọng, vì các khóa trên thực tế là các chuỗi ký tự và dự kiến sẽ được chuyển đổi thành một số trước khi hoạt động phần còn lại của mô-đun để đảm bảo phân phối đồng đều hơn ở các bit thấp hơn. Ưu điểm của việc sử dụng bảng băm có kích thước này là khả năng sử dụng thao tác nhanh hơn - bitwise "và", thay vì chia số nguyên với phần dư. Bước s ($s > 2$) mà chúng ta sẽ thực hiện phép thử sẽ là một số nguyên tố: do đó chúng ta đảm bảo rằng thuộc tính n và s thoả mãn ràng chúng là nguyên tố cùng nhau.
- Nếu kích thước được xác định trước cho bảng băm chứng tỏ không đủ, chúng ta sẽ cấp phát bộ nhớ cho nhiều phần tử hơn:

chúng ta sẽ tăng gấp đôi kích thước của nó và do đó giữ thuộc tính dung lượng của nó là lũy thừa 2.

Chương trình 2.6. Bảng băm tập hợp (206hashset.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define S 107 /* bước phóng đại xung đột */
#define MAX_FILL_LEVEL 0.8 /* Mức lấp đầy tối đa */
unsigned long n = 32; /* kích thước ban đầu của bảng băm */

struct singleWord {
    char *word; /* từ khóa - chuỗi ký tự */
    unsigned long freq; /* tần suất xuất hiện của từ */
} *ht;
unsigned long count;

/* Hàm băm chuỗi ký tự */
unsigned long hashFunction(const char *key)
{ unsigned long result = 0;
    while (*key)
        result += result + (unsigned char) *key++;
    return result & (n - 1); }

/* Khởi tạo bảng băm */
void initHashtable(void)
{ unsigned long i;
    count = 0;
    ht = (struct singleWord *) malloc(sizeof(*ht)*n);
    for (i = 0; i < n; i++)
        ht[i].word = NULL;
}

/* Tìm kiếm trong bảng băm: trả về 1 nếu thành công và 0 - nếu không */
/* Khi thành công: * ind chứa chỉ số của phần tử được tìm thấy */
/* Không thành công: vị trí trống, nơi có thể lấp vào */
char get(const char *key, unsigned long *ind)
{ unsigned long k;
    *ind = hashFunction(key);
    k = *ind;
```

```

do {
    if (NULL == ht[*ind].word) return 0;
    if (0 == strcmp(key, ht[*ind].word)) return 1;
    *ind = (*ind + S) & (n - 1);
} while (*ind != k);
return 0;
}

/* Mở rộng bảng băm*/
void resize(void)
{
    unsigned long ind, hashInd;
    struct singleWord *oldHashTable;

/* 1. Lưu thư mục vào bảng băm */
oldHashTable = ht;

/* 2. Mở rộng gấp đôi */
n <= 1;

/* 3. Phân bổ bộ nhớ cho bảng băm mới*/
ht = (struct singleWord *) malloc(sizeof(*ht)*n);
for (ind = 0; ind < n; ind++)
    ht[ind].word = NULL;

/*4. Di chuyển các bản ghi sang bảng băm mới*/
for (ind = 0; ind < (n >> 1); ind++) {
    if (oldHashTable[ind].word != NULL) {
        /* Di chuyển bản ghi đến vị trí mới*/
        if (!get(oldHashTable[ind].word, &hashInd))
            ht[hashInd] = oldHashTable[ind];
        else
            printf("Lỗi phần mở rộng bảng băm!\n");
    }
}

/* 5. Giải phóng bộ nhớ cũ */
free(oldHashTable);
}

/* Thêm một phần tử vào bảng băm */

```

```

void put(char *key)
{ unsigned long ind;
if (!get(key, &ind)) { /* Từ không có trong bảng băm */
    ht[ind].word = strdup(key);
    ht[ind].freq = 1;
    if (++count > ((unsigned long) n * MAX_FILL_LEVEL)) resize();
}
else ht[ind].freq++;

/* In bảng băm */
void printAll(void)
{ unsigned long ind;
for (ind = 0; ind < n; ind++)
    if (ht[ind].word != NULL)
        printf("%s %ld \n", ht[ind].word, ht[ind].freq);
}

/* Hủy bảng băm */
void destroy(void)
{ unsigned long ind;
for (ind = 0; ind < n; ind++)
    if (ht[ind].word != NULL) free(ht[ind].word);
    free(ht);
}

int main()
{
    unsigned long find;
    initHashtable();
    put("reload");
    put("crush tour");
    put("room service");
    put("load");
    put("reload");
    put("reload");

    printAll();

    if (get("reload", &find))
        printf("Tần suất từ 'reload': %d \n", ht[find].freq);
    else
}

```

```

printf("Thiếu từ 'reload'!");

if (get("download", &find))
    printf("Tần suất từ 'download': %d \n", ht[find].freq);
else
    printf("Thiếu từ 'download'!");
destroy();
return 0;
}

```

Kết quả thực hiện chương trình:

```

load 1
reload 3
crush tour 1
room service 1
Tần suất của từ 'reload': 3
Thiếu từ 'download'!

```

Bài tập

- ▷ 2.39. Để triển khai một hàm để loại trừ một phần tử khỏi bảng băm bằng một khóa. Độ phức tạp phải không đổi. Lưu ý rằng không đủ nếu chỉ tìm phần tử có khóa được chỉ định và xóa nó, như trong quá trình triển khai với danh sách tràn: nghĩa là điều này sẽ phá vỡ chuỗi các phần tử có cùng mã băm. của chuỗi các từ đồng nghĩa.
- ▷ 2.40. Có thể sửa đổi thuật toán mở rộng bảng băm từ 2.4.3 không. (thực hiện nguyên tắc kiểm tra tuyến tính) để không cần phải di chuyển từng phần tử (làm nóng lại) mà chỉ cần sao chép bộ nhớ?
- ▷ 2.41. Thực hiện một phương pháp để "thu nhỏ" bảng băm: giảm kích thước của nó khi số lượng phần tử của nó giảm. Điều gì có thể là tiêu chí để giảm kích thước của bảng băm? Nên tăng hoặc giảm tốc sau khi đàm nén? Tại sao?
- ▷ 2.42. Trong chương trình đề xuất ở trên, không được phép điền bảng băm trên 80%: khi đạt đến mức này, nó sẽ tự động mở rộng. Các nghiên cứu chỉ ra rằng với mức độ lấp đầy như vậy và một hàm băm được chọn đúng cách, số lượng mẫu trung bình nhỏ hơn 2. Để

đánh giá thực nghiệm mỗi quan hệ giữa số lượng mẫu trung bình và mức độ lấp đầy của bảng băm. Số lượng mẫu trung bình tại thời điểm điền đầy là bao nhiêu: 85%, 90%, 93%, 95%, 97%, 98%, 99%, 100%?

▷ 2.43. Trong cách triển khai được đề xuất ở trên, giá trị của hàm băm có tính đến vị trí của các ký hiệu:

```
result += result + (unsigned char) * key ++;
```

Để so sánh với hàm băm cổ điển cho các chuỗi ký tự:

```
result = result + (unsigned char) * key ++;
```

Ước tính số lượng mẫu trung bình cần thiết để phát hiện một phần tử có khóa hoặc xác định rằng phần tử đó bị thiếu trong mỗi một trong hai hàm băm.

▷ 2.44. Trong cách triển khai được đề xuất ở trên, chúng ta đã sử dụng băm đóng với các mẫu tuyển tính. Thử nghiệm với các chiến lược quản lý xung đột tiêu chuẩn khác như lấy mẫu bậc hai và băm kép với các chức năng được chọn phù hợp. Số lượng mẫu trung bình giảm hay tăng? Tỷ lệ giữa số lượng mẫu trung bình trong các chiến lược quản lý xung đột khác nhau có phụ thuộc vào mức độ hoàn thành của bảng băm không? Và kích thước của nó? Lớp 2 có phải là lựa chọn tốt?

▷ 2.45. Để kiểm tra việc triển khai được đề xuất ở trên trên các dữ liệu đầu vào khác nhau. Số lượng mẫu trung bình có thay đổi trong quá trình tìm kiếm không?

2.8. Câu hỏi và bài tập

▷ 2.46. *Danh sách liên kết đôi*

Để biên dịch triển khai động của danh sách tuyển tính hai liên kết. Một danh sách tuyển tính liên kết đôi được định nghĩa một cách đơn giản như sau:

```
typedef int data;
typedef int keyType;

struct list {
    keyType key;
```

```

data info;
struct list *prev;
struct list *next;
};

```

Mỗi phần tử trong danh sách liên kết đôi có hai con trỏ - đến phần tử tiếp theo (như trong danh sách liên kết đơn) và đến phần tử trước (con trỏ trước prev).

Hãy hiện thực các thao tác cơ bản để làm việc với danh sách tuyến tính - bao gồm một phần tử (trước và sau một phần tử được chỉ định bởi con trỏ), cũng như để thực hiện (với độ phức tạp không đổi), thao tác xóa một phần tử được chỉ ra bởi một con trỏ.

► 2.47. Danh sách tuần hoàn

Danh sách liên kết đơn theo chu kỳ được gọi là danh sách liên kết đơn tuyến tính, trong đó con trỏ đến mục tiếp theo bên cạnh mục cuối cùng trỏ đến mục đầu tiên trong danh sách.

Hãy thiết lập triển khai động của danh sách liên kết đơn theo chu kỳ và triển khai các hoạt động cơ bản để làm việc với danh sách đó.

► 2.48. Hàng đợi ưu tiên

Hàng đợi ưu tiên tìm thấy nhiều ứng dụng (bộ nhớ đệm, v.v.). Mỗi phần tử, ngoài dữ liệu mà nó chứa, được đặc trưng bởi một số nguyên - "mức độ ưu tiên" của nó. Khi một phần tử được đưa vào hàng đợi, điều này xảy ra không phải ở cuối mà ngay sau phần tử cuối cùng có mức ưu tiên cao hơn phần tử của nó (nếu các phần tử bằng nhau, thứ tự theo thứ tự nhận). Do đó, tất cả các phần tử có mức độ ưu tiên cao hơn nằm trước nó và các phần tử có mức độ ưu tiên thấp hơn - sau nó trong hàng đợi. Việc loại trừ khỏi hàng đợi được thực hiện ngay từ đầu - luôn có phần tử có mức ưu tiên cao nhất.

Viết chương trình mô phỏng hàng đợi ưu tiên và thực hiện các thao tác bật và tắt cơ bản của bản ghi độ phức tạp ($\log 2 n$) [Nakov-1998] [Wirth-1980].

Lưu ý: Cấu trúc dữ liệu kim tự tháp được thảo luận trong 3.1.9. "Phân loại kim tự tháp của Williams." Ngoài việc triển khai hiệu quả hàng đợi ưu tiên, kim tự tháp còn được sử dụng để giảm độ phức

tập trong nhiều thuật toán khác - thuật toán Dijkstra (xem 5.4.2.), V.v.

► 2.49. Hàng đợi "nhóm"

Một loại đuôi ít được biết đến, thường thấy trong cuộc sống hàng ngày, được gọi là đuôi hàng đợi "đội". Ví dụ, hàng đợi hình thành trước ghế học sinh là hàng đợi "đội". Khi một mục mới tham gia vào hàng đợi, trước tiên nó sẽ tìm kiếm "người quen" của mình trong đó và nếu được tìm thấy, sẽ tham gia ngay sau họ. Nếu anh ta không tìm thấy nó, nguyên tố mới sẽ không còn may mắn và sẽ xếp hàng. Loại trừ một phần tử khỏi hàng đợi là cách tiêu chuẩn - chỉ từ đầu hàng đợi. Viết chương trình mô phỏng hàng đợi "nhóm" cho một số nhóm phần tử nhất định (các phần tử thuộc một nhóm được coi là "đã biết").

► 2.50. Đảo ngược bản ghi trường

Viết chương trình tính giá trị của một biểu thức được viết bằng ký hiệu Ba Lan ngược. Để giải quyết vấn đề sử dụng một ngăn xếp [Nakov-1998].

► 2.51. Khôi phục biểu thức

Viết chương trình khôi phục ký hiệu đại số cổ điển của biểu thức có dấu ngoặc từ một biểu thức số học đã cho được viết bằng ký hiệu trường ngược. Để giải quyết vấn đề theo hai cách: có và không sử dụng gỗ nhị phân [Nakov-1998].

► 2.52. Phân tích cú pháp nhanh

Viết chương trình đọc mã nguồn của chương trình C và kiểm tra xem (các) cặp { }; { i } ; [i] ; /* i */ tham gia một cách đối xứng trong đó.

Gợi ý: Sử dụng ngăn xếp.

► 2.53. Mô phỏng đệ quy

Khi các cuộc gọi đệ quy được thực hiện, dữ liệu cục bộ cho mỗi cuộc gọi kế tiếp được lưu trữ trong bộ nhớ ngăn xếp (ở cấp trình biên dịch / hệ điều hành, quá trình này như sau: có một phần bộ nhớ được gọi là ngăn xếp chương trình và một thanh ghi con trỏ lên trên cùng của ngăn xếp).

Mỗi hàm đệ quy có thể được thay thế bằng một biến thể lặp tương đương. Điều này có thể được thực hiện bằng một trong những cách sau:

- Tất cả các tham số của hàm đệ quy được xuất dưới dạng biến toàn cục và được sửa đổi cho phù hợp. Cách tiếp cận này liên quan đến việc sửa đổi logic của mã chương trình và không phải lúc nào cũng có thể thực hiện được. (Tại sao?)
- Mô phỏng tổ chức ngăn xếp chương trình: Sử dụng ngăn xếp của riêng bạn, ngăn xếp này ghi lại các giá trị cuối cùng của tất cả các biến liên quan đến hàm. Đến lượt nó, hàm hoạt động với tập biến cuối cùng được lưu trữ trong ngăn xếp.

Để thực hành kỹ thuật thứ hai, hãy giải quyết các tác vụ sau bằng cách sử dụng ngăn xếp:

- Nâng một số thực lên lũy thừa.
- Tìm số Fibonacci thứ n. Để duy trì tính logic và tính không hiệu quả của việc thực hiện đệ quy trực tiếp, xem 1.2.2.
- Bài toán của Hanoi Towers (xem 7.8.).

► 2.54. Đa thức

Để viết chương trình làm việc với đa thức với hệ số thực và nghiệm nguyên có dạng:

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0,$$

được trình bày thông qua một danh sách liên kết. Mô-đun để làm việc với đa thức phải bao gồm các phép toán cộng, trừ, nhân, chia đa thức, cũng như tìm giá trị của đa thức cho một giá trị nhất định của biến chưa biết x [Nakov-1998].

► 2.55. Sắp xếp theo cây

Trong 2.3. chúng ta đã chỉ ra cách có thể lấy được các khóa của cây tìm kiếm nhị phân, được sắp xếp theo thứ tự tăng dần. Điều gì cần thay đổi khi thu thập dữ liệu để chuỗi khóa được sắp xếp theo thứ tự giảm dần?

► 2.56. Số lượng

Viết chương trình tìm tổng các mức ở tất cả các đỉnh trong cây nhị phân.

► 2.57. *Gắn như một chiếc lá*

Viết chương trình in tất cả các ngọn của cây nhị phân chỉ có các lá là cây thừa kế.

► 2.58. *Phục hồi cây*

Kết quả của hai lần thu thập thông tin cây nhị phân giữa {LCD, LDC và CLD} được đưa ra. Để tìm lần thu thập thông tin thứ ba [Nakov-1998].

► 2.59. *Xây dựng một cái cây cân bằng hoàn hảo*

Viết chương trình xây dựng một cây cân bằng hoàn hảo trên một cây chứa các khóa giống nhau.

Gợi ý: Một danh sách tuyến tính có thể được xây dựng trong đó các phần tử tham gia theo thứ tự tăng dần, và sau đó được sử dụng để xây dựng một cây cân bằng hoàn hảo [Nakov-1998].

► 2.60. *Kiểm tra cây cân đối hoàn hảo*

Viết hàm kiểm tra xem cây tìm kiếm nhị phân có cân bằng hoàn hảo hay không.

► 2.61. *Tìm kiếm trong không gian*

Một cây tìm kiếm nhị phân và hai số nguyên t_1 và t_2 được đưa ra. Tìm tất cả các đỉnh x của cây có khóa nằm trong khoảng $[t_1, t_2]$, tức là $t_1 \leq \text{key}[x] \leq t_2$.

CHƯƠNG 3

SẮP XẾP DỮ LIỆU VÀ THUẬT TOÁN

3.1. Giới thiệu	232
3.2. Sắp xếp theo so sánh	234
3.2.1. Cây so sánh	234
3.2.2. Sắp xếp theo cách chèn	236
3.2.3. Sắp xếp theo thứ tự chèn với bước giảm dần. Thuật toán của Shell	240
3.2.4. Phương pháp bong bóng	243
3.2.5. Sắp xếp bằng cách lắc	245
3.2.6. Nhanh chóng phân loại Hoar	246
3.2.7. Phương pháp Thỏ và Rùa	256
3.2.8. Sắp xếp theo lựa chọn trực tiếp	259
3.2.9. Sắp xếp kim tự tháp Williams	260
3.2.10. Độ phức tạp về thời gian tối thiểu của việc sắp xếp theo cách so sánh	266
3.2.11. Bài tập	268
3.3. Sắp xếp theo sự biến đổi	268
3.3.1. Sắp xếp theo tập hợp	268
3.3.2. Sắp xếp theo số đếm	272
3.3.3. Sắp xếp theo bit	276
3.3.4. Phương pháp hệ đếm số	282
3.3.5. Sắp xếp theo hoán vị	286
3.4. Sắp xếp song song	288
3.4.1. Nguyên tắc về số không và số một	292
3.4.2. Trình tự bitonic	294
3.4.3. "Rõ ràng một nửa"	295
3.4.4. Sắp xếp chuỗi bitonic	295
3.4.5. Sắp xếp sơ đồ hợp nhất	296
3.4.6. Sắp xếp sơ đồ phân loại	296
3.4.7. Sơ đồ phân loại chuyển vị	297
3.4.8. Sơ đồ sáp nhập Batcher chẵn lẻ	298
3.4.9. Lược đồ sắp xếp chẵn-lẻ	298

3.4.10. Lược đồ hoán vị	299
3.5. Câu hỏi và bài tập	301

"... có nhiều phương pháp tốt nhất, tùy thuộc vào cái gì sẽ được sắp xếp, trên máy gì, cho mục đích gì."

D.E.Knuth

3.1. Giới thiệu

Thông thường, khi làm việc với dữ liệu lớn cùng loại, cần phải đưa ra một sắc lệnh để tạo điều kiện thuận lợi cho việc xử lý chúng. Như chúng ta sẽ thấy trong Chương 4, việc sắp xếp các phần tử có thể cho chúng ta một thuật toán tìm kiếm hiệu quả hơn nhiều so với khi dữ liệu không được sắp xếp.

Người ta chấp nhận rằng quá trình sắp xếp lại (hoán vị theo một cách thích hợp) các phần tử của một tập hợp các đối tượng theo một thứ tự nhất định được gọi là *sắp xếp*. Sắp xếp là một hoạt động chính với lĩnh vực ứng dụng rộng rãi: từ điển, danh bạ điện thoại, mục lục tài liệu tham khảo và nói chung ở mọi nơi cần tìm kiếm nhanh chóng và tìm thấy nhiều đối tượng khác nhau. Sắp xếp là một phần không thể thiếu trong cuộc sống hàng ngày của chúng ta - chẳng sau mọi sự sắp xếp, từ bàn làm việc chúng ta ngồi, đến tủ quần áo, túi xách, v.v. có một số kiểu sắp xếp.

Sắp xếp là một khái niệm cực kỳ rộng và, tùy thuộc vào loại dữ liệu được sắp xếp, có thể được thực hiện theo nhiều cách khác nhau. Sự đa dạng của các thuật toán đến nỗi Knuth dành toàn bộ tập thứ ba (hơn 800 trang) cho chuyên khảo nổi tiếng của mình *Nghệ thuật lập trình máy tính* (xem Knuth-3/1968). Nicklaus Wirth, người tạo ra ngôn ngữ Pascal, cũng rất chú ý đến chúng trong "*Thuật toán + Cấu trúc dữ liệu = Chương trình*" (xem [Wirth-1980]).

Có nhiều cách phân loại khác nhau của các thuật toán sắp xếp. Có lẽ phổ biến nhất là tùy thuộc vào vị trí của dữ liệu. Dựa trên tiêu chí này, chúng ta phân biệt giữa nội bộ (dữ liệu nằm trong RAM của máy tính và thường có thể truy cập trực tiếp vào bất kỳ phần tử nào của tập hợp) và bên ngoài (dữ liệu nằm trong bộ nhớ ngoài

của máy tính và quyền truy cập thường nhất quan nghiêm ngặt, bắt đầu từ phần tử đầu tiên). Tùy thuộc vào phép toán được thực hiện trên các phần tử, chúng ta phân biệt giữa sắp xếp bằng cách so sánh (thường xuyên nhất với sự trợ giúp của $<$, $>$ và $=$) và bằng phép biến đổi (với sự trợ giúp của các phép toán số học, không so sánh trực tiếp các cặp phần tử). Các phân loại quan trọng khác dựa trên các thuộc tính nhất định của các thuật toán sắp xếp. Ví dụ, ổn định và không ổn định. Một phương thức được gọi là ổn định nếu thứ tự tương đối của các phần tử có khóa bằng nhau không thay đổi trong quá trình sắp xếp. Phương pháp sắp xếp ổn định được ưu tiên khi các phần tử của tập hợp đã được sắp xếp theo một số tiêu chí khác. Ví dụ: chúng ta muốn sắp xếp hệ thống tệp của mình theo tên và phần mở rộng: các tệp được sắp xếp theo tên và những tệp có cùng tên được sắp xếp theo phần mở rộng. Điều này có thể được thực hiện theo hai bước: 1) sắp xếp theo phần mở rộng và 2) sắp xếp theo tên. Trong khi ở loại đầu tiên (theo phần mở rộng), tính ổn định của thuật toán là không liên quan, ở loại thứ hai (theo tên), việc sử dụng một phương pháp ổn định bây giờ sẽ là bắt buộc.

Quá trình sắp xếp các phần tử từ một tập hợp, như đã đề cập ở trên, thường được rút gọn để sắp xếp lại chúng. Chúng ta sẽ định nghĩa các khái niệm một cách chặt chẽ hơn. Cho tập hợp M với các phần tử đã cho

$$a_1, a_2, \dots, a_n,$$

và hàm f xác định trên chúng.

Bằng cách sắp xếp các phần tử của M , chúng ta có nghĩa là hoán vị của chúng theo thứ tự thích hợp

$$a_{i_1}, a_{i_2}, \dots, a_{i_n}$$

để nó được hoàn thành:

$$f(a_{i_1}) \leq f(a_{i_2}) \leq \dots \leq f(a_{i_n}).$$

Hàm f được gọi là *hàm sắp xếp* của tập hợp. Chúng ta sẽ xem xét rằng nó được tính toán trước cho từng phần tử và được ghi nhớ rõ ràng như một phần (trường, *khóa*) của nó. Các phần tử của tập hợp được sắp xếp thường được biểu diễn dưới dạng bản ghi, một trong các trường là khóa:

```
#define MAX 100
struct CElem {
    int key;
    /* ..... */
    Một số dữ liệu
    ..... *
} m [MAX];
```

Đối với các phương pháp sắp xếp phổ biến, và nếu có thể, chúng ta sẽ thực hiện theo khai báo ở trên về loại phần tử của tập hợp. Chúng ta thường sẽ biểu diễn tập hợp dưới dạng một mảng, nhưng đôi khi chúng ta cũng sẽ sử dụng biểu diễn động dưới dạng danh sách được liên kết.

Yêu cầu chính đối với các thuật toán sắp xếp là chi phí tối thiểu của bộ nhớ bổ sung. Một yêu cầu quan trọng khác là số lượng tối thiểu so sánh và trao đổi các yếu tố. Việc sắp xếp thường được thực hiện bằng cách hoán đổi đơn giản hai phần tử của mảng. Trong các chương trình sau để trao đổi giá trị của hai biến, chúng ta sẽ sử dụng hàm sau:

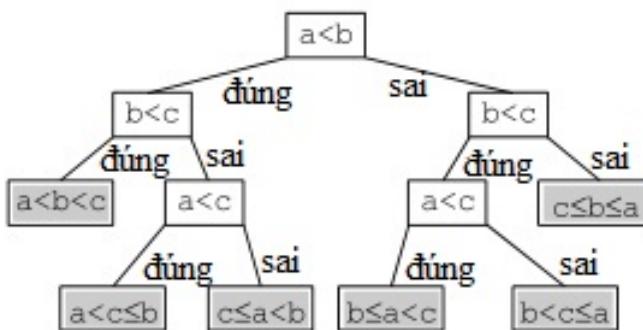
```
/* Hoán đổi các giá trị của *x1 và *x2 */
void swap(struct CElem *x1, struct CElem *x2)
{ struct CElem tmp = *x1; *x1 = *x2; *x2 = tmp; }
```

3.2. Sắp xếp theo so sánh

Tất cả các thuật toán trong đoạn này đều thuộc loại thuật toán sắp xếp so sánh. Đây là các thuật toán cổ điển trong đó phép toán duy nhất được phép là so sánh giữa các cặp phần tử sử dụng các phép toán $<$ (\leq), $>$ (\geq) và $=$ (\neq).

3.2.1. Cây so sánh

Đây có lẽ là phương pháp cơ bản nhất để phân loại theo phương pháp so sánh. Nó dựa trên một loạt các so sánh, mỗi so sánh mang lại cho chúng ta thông tin bổ sung. Quá trình tiếp tục cho đến khi tập hợp được sắp xếp hoàn toàn. Tùy thuộc vào kết quả của phép so sánh, chúng ta nhận được các tỷ lệ khác nhau giữa các yếu tố và do

Hình 3.1. Cây so sánh cho tập $\{a, b, c\}$.

đó - sự tiếp tục khác nhau của quá trình. Chúng ta có thể giả định rằng bất kỳ so sánh nào của dạng $x < y$ đều có hai kết quả: có, nếu x nhỏ hơn y và không - nếu không. Quá trình so sánh có thể được biểu diễn bằng đồ thị dưới dạng cây nhị phân, trong danh sách chứa các trình tự được sắp xếp và trong danh sách không - so sánh giữa các cặp phần tử của tập hợp. Hình 3.1 minh họa phương thức trên tập ba phần tử $\{a, b, c\}$.

Cây cung cấp cho chúng ta thuật toán sau để sắp xếp tập hợp:

```

if (a < b)
  if (b < c) printf("a,b,c");
  else
    if (a < c) printf("a,c,b");
    else printf("c,a,b");
else
  if (b < c)
    if (a < c) printf("b,a,c");
    else printf("b,c,a");
  else printf("c,b,a");
  
```

Thuật toán rất hay và hữu ích trong điều kiện nó 1) đảm bảo cho chúng ta số lượng phép so sánh tối thiểu (Tại sao?), Và 2) đưa ra một cây so sánh rõ ràng (thuật toán thứ hai được sử dụng để chứng minh một kết quả quan trọng: độ phức tạp thời gian tối thiểu của bất kỳ thuật toán nào để phân loại bằng cách so sánh - xem 3.2.10). Ưu điểm của nó, thật không may, kết thúc ở đó. Không khó để nhận

thấy rằng số lượng đầu ra có thể có (lá cây) là chương trình $n!$. Cần tìm kiếm các phương pháp sắp xếp khác hiệu quả hơn.

Bài tập

- ▷ 3.1. Chứng minh rằng cây so sánh đảm bảo số lượng so sánh tối thiểu trong trường hợp xấu nhất.

3.2.2. Sắp xếp theo cách chèn

Có ba phương pháp phổ thông cơ bản cổ điển để sắp xếp theo cách so sánh: theo chèn, theo lựa chọn và theo phương pháp bong bóng. Trọng tâm của mỗi chúng là một ý tưởng đơn giản cho phép thực hiện nhanh chóng và rõ ràng. Phương pháp sắp xếp cơ bản có hiệu quả với số lượng phần tử tương đối nhỏ (khoảng 20 phần tử) và thường được sử dụng trong thực tế. Thật không may, với số lượng phần tử lớn hơn, tốc độ của chúng giảm mạnh, điều này khiến bạn cần phải sử dụng các phương pháp khác. Thực vậy, cả ba phương pháp cơ bản đều được đặc trưng bởi độ phức tạp thuật toán $\Theta(n^2)$, chậm hơn nhiều so với độ phức tạp $\Theta(n \log_2 n)$, đây là đặc điểm của các phương pháp sắp xếp hiện đại như phương pháp hình chóp (xem 3.2.9) hoặc sắp xếp nhanh (xem 3.2.6). Tuy nhiên, các phương pháp cơ bản có vị trí của chúng, vì đối với các trình tự đủ nhỏ, chúng hiệu quả hơn và như chúng ta sẽ thấy bên dưới, thường được sử dụng trong các biến thể lai để tăng tốc độ. Ví dụ, một cách tiếp cận sắp xếp nhanh được sử dụng rộng rãi là sử dụng một phương pháp đơn giản hơn, chẳng hạn như sắp xếp chèn, khi đến một phân vùng có ít phần tử.

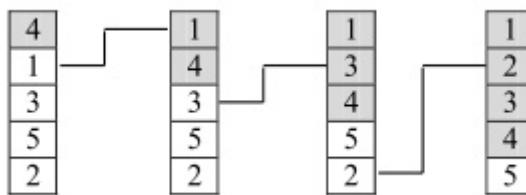
Sắp xếp chèn là một phương pháp sắp xếp thứ tự nổi tiếng, trong đó người chơi cầm quân bài ở tay trái, lấy từng quân bài ở tay phải và đặt chúng vào đúng vị trí. Để cắm thẻ vào đúng vị trí, cần phải so sánh nhất quán (bằng cách nhìn nhanh) với các thẻ đã được sắp xếp cho đến khi tìm được vị trí chính xác.

Hãy quay trở lại thuật ngữ của chúng ta. Chúng ta có một mảng các phần tử **struct** CElem với một khóa key mà chúng ta muốn sắp xếp. Mảng được chia thành khu vực được sắp xếp và không được sắp xếp. Vùng đã sắp xếp thường nằm ở đầu mảng, ban đầu chỉ bao gồm phần tử đầu tiên của nó. Việc sắp xếp diễn ra trong $n - 1$ bước.

Trong bước thứ i , khu vực đã sắp xếp được mở rộng bởi một phần tử ở bên phải và cho mục đích này ($i + 1$) phần tử đầu tiên (giả sử ký hiệu là x) được chèn vào một vị trí thích hợp trong chuỗi đã sắp xếp, tức là trong số các phần tử từ 1 đến i .

Việc chèn được thực hiện như thế nào? Một thuật toán hiển nhiên là so sánh tuần tự và có thể trao đổi x với phần tử bên trái của nó. Quá trình tiếp tục cho đến khi một trong các tình huống sau xảy ra (xem Hình 3.2):

- 1) đến phần tử có khóa nhỏ hơn hoặc bằng khóa của x ;
- 2) đến phần tử đầu tiên của mảng.



Hình 3.2. Sắp xếp theo cách chèn.

Sau đây là một ví dụ thực hiện. Lưu ý rằng phần tử đầu tiên của mảng thực sự là $m[0]$

Chương trình 3.1. Sắp xếp theo cách chèn (301insert_s.c)

```
void straightInsertion (struct CElem m[], unsigned n)
{ struct CElem x;
  unsigned i;
  int j;
  for (i = 0; i < n; i++) {
    x = m[i];
    j = i - 1;
    while (j >= 0 && x.key < m[j].key)
      m[j+1] = m[j--];
    m[j+1] = x;
  }
}
```

Tất cả các kiểm tra có thể được hợp nhất thành một bằng cách thêm một phần tử khóa $-\infty$ làm phần tử 0 của mảng. Phương pháp

này được gọi là *phương pháp giới hạn*. Vì giá trị của nó khó biểu diễn trong hệ thống máy tính, nên việc sử dụng một khóa từ tập hợp các khóa hợp lệ làm giới hạn sẽ dễ dàng hơn. Không khó để coi rằng giá trị của khóa x sẽ thực hiện một công việc tuyệt vời. (Tại sao?) Chúng ta nhận được phần mềm triển khai thuật toán chèn trực tiếp sau (Lần này các phần tử nằm ở vị trí 1, 2, ..., n):

Chương trình 3.2. Sắp xếp theo cách chèn (302inserts2.c)

```
void straightInsertion (struct CElem m[], unsigned n)
{ unsigned i, j;
struct CElem x;
for (i = 1; i <= n; i++) {
    x = m[i]; m[0].key = x.key;
    for (j = i - 1; x.key < m[j].key; j--)
        m[j+1] = m[j];
    m[j+1] = x;
}
}
```

Cách chèn	Số tối thiểu	Số trung bình	Số tối đa
So sánh	$n - 1$	$(n^2 + n - 2)/4$	$(n^2 + n)/2 - 1$
Trao đổi	$2(n - 1)$	$(n^2 + 9n - 10)/4$	$(n^2 + 3n - 4)/2$

Bảng 3.1. So sánh và trao đổi trong việc sắp xếp theo cách chèn.

Không khó để xác định rằng số lượng so sánh trung bình được thực hiện, cũng như số lượng trao đổi được yêu cầu (lưu ý rằng trong chương trình trên hầu như không có các trao đổi trực tiếp, thay vào đó sử dụng các *phép gán một chiều*), là bậc n^2 . Điều này cũng có thể được nhìn thấy từ Bảng ?? đính kèm. Người đọc tò mò có thể tìm thấy lời giải thích chi tiết cho những đánh giá trên trong [Knuth-3/1968] [Wirt-1980].

Chúng ta có thể tăng hiệu quả của việc phân loại chèn không? Và có, và không. Thật vậy, bằng cách sử dụng tìm kiếm nhị phân (xem 4.3.), Chúng ta có thể giảm số lượng *phép so sánh* cần thiết để tìm vị trí chèn x từ bước thứ i, thành $\log_2 i$:

Chương trình 3.3. Sắp xếp theo cách chèn (303insert_b.c)

```

void binaryInsertion(struct CElem m[], unsigned n)
{
    struct CElem x;
    unsigned i, med, r;
    int j, l;
    for (i = 1; i < n; i++) {
        x = m[i];
        l = 0;
        r = i - 1;
        /* Tìm kiếm nhị phân*/
        while (l <= (int)r) {
            med = (l + r) / 2;
            if (x.key < m[med].key)
                r = med - 1;
            else
                l = med + 1;
        }
        /* Nơi đã được tìm thấy. Sau đó chèn và di chuyển sang bên phải
        */
        for (j = i - 1; j >= l; j--)
            m[j + 1] = m[j];
        m[l] = x;
    }
}

```

Thật không may, điều sau sẽ không ảnh hưởng đến số lượng trao đổi, bởi vì, mặc dù chúng ta biết vị trí cụ thể của phần chèn, nhưng chúng ta phải xem qua các phần tử của mảng cho đến khi nó đạt đến. Sự bất tiện này có thể được khắc phục bằng cách sử dụng danh sách được liên kết động, nhưng sau đó chúng ta không thể áp dụng tìm kiếm nhị phân.

Bài tập

▷ 3.2. Hãy chứng minh rằng khóa của phần tử được chèn là một giới hạn tốt khi sử dụng phương pháp của giới hạn để thực hiện sắp xếp bằng cách chèn.

▷ 3.3. Hãy so sánh các biến thể khác nhau của sắp xếp theo cách chèn.

► 3.4. Tính toán công thức từ Bảng 3.1.

3.2.3. Sắp xếp theo thứ tự chèn với bước giảm dần. Thuật toán của Shell

Năm 1959, Shell đề xuất một cải tiến tốt về phương pháp sắp xếp chèn. Ý tưởng là thực hiện nhiều sắp xếp bằng cách chèn trực tiếp một phần của các phần tử của mảng với bước δ , bước này giảm dần và cuối cùng đạt đến 1.

Quá trình sắp xếp bằng cách chỉ chèn trực tiếp các tập con có dạng $x_k, x_{k+\delta}, x_{k+2\delta}, \dots (1 \leq k \leq \delta)$ sẽ được gọi là δ -sắp xếp. Cho dãy các bước $\delta_s, \delta_{s-1}, \dots, \delta_1$, thỏa mãn điều kiện

$$\delta_s > \delta_{s-1} > \dots > \delta_1 = 1.$$

Rõ ràng, do ứng dụng tuần tự của δ_s -sắp xếp, δ_{s-1} -sắp xếp, ..., δ_1 -sắp xếp tập hợp đầu ra sẽ được sắp xếp. Không khó để thấy rằng nếu một dãy đã được δ -sắp xếp, thì sau khi áp dụng δ' -sắp xếp ($\delta' < \delta$), nó tiếp tục được (-sắp xếp). (Tại sao?) Như vậy, mỗi lần sắp xếp tiếp theo sẽ tận dụng lợi thế của lần sắp xếp trước đó và cuối cùng việc sắp xếp theo bước 1 sẽ hoàn thành phần còn lại của công việc.

Mỗi lần sắp xếp được giảm xuống δ số lượng kiểu bằng cách chèn trực tiếp, mỗi kiểu cần có giới hạn riêng. Thêm một phần tử 0 ở đầu mảng là không đủ. Chúng ta cần tổng số δ_s trong số lượng dấu phân cách, điều này yêu cầu mở rộng mảng thêm δ_s về số lượng phần tử. Chúng ta sẽ làm việc với một mảng duy nhất và chúng ta sẽ sử dụng các phần tử đầu tiên của nó làm giới hạn. Để có thể tham chiếu đến các phần tử có chỉ số âm, chúng ta sẽ sử dụng một mẹo: chúng ta sẽ gọi hàm shellSort() không phải với shellSort(m, MAX), mà với shellSort(m + step0 + 1, MAX) .

Chương trình 3.4. Thuật toán shell (304shell.c)

```
#define MAX 100
#define STEPS_CNT 4
#define steps0 40
const unsigned steps[STEPS_CNT] = { steps0, 13, 4, 1 };
struct CElem {
    int key;
```

```

/*
    .....
    Một số dữ liệu
    .....
};

.....
void shellSort(struct CElem m[], unsigned n)
{ int i,j,k,s;
  unsigned stepInd;
  struct CElem x;

  for (stepInd = 0; stepInd < STEPS_CNT; stepInd++) {
    s = -(k = steps[stepInd]); /*Giới hạn*/
    for (i = k + 1; i <= (int)n; i++) {
      x = m[i];
      j = i - k;
      if (0 == s)
        s = -k;
      m[++s] = x;
      while (x.key < m[j].key) {
        m[j + k] = m[j];
        j -= k;
      }
      m[j + k] = x;
    }
  }
}

int main()
{
  struct CElem m[MAX + steps0 + 2];
  .....
  shellSort(m + steps0 + 1, MAX);
  .....
  return 0;
}

```

Ý tưởng về dấu phân cách là tốt cho việc sắp xếp chèn, nhưng ở đây cần có một số dấu phân cách, điều này không hiệu quả và dẫn đến sự bất tiện trong việc định địa chỉ mảng, mà chúng ta đã giải quyết ở trên bằng "chương trình nắm bắt". Nó có thể hiệu quả và mang tính hướng dẫn, nhưng nó có cần thiết không? Nếu chúng ta

suy nghĩ một chút, chúng ta dễ dàng đi đến kết luận rằng chúng ta thực sự có thể làm mà không có giới hạn!

Kết quả là, chúng ta nhận được một đoạn mã ngắn hơn và rõ ràng hơn nhiều. Tuy nhiên, chúng ta lưu ý rằng chúng ta "trả tiền" cho điều này một cách hiệu quả, vì thuật toán mới của chúng ta (xem bên dưới) thực hiện nhiều gấp đôi so sánh trong vòng lặp while nội bộ của nó. Ở đây chúng ta sử dụng một dãy 16 phần tử hiệu quả khác (chúng ta sắp xếp các phần tử từ $l + 1$ đến r).

Chương trình 3.5. Thuật toán shell (305shell2.c)

```

void shellSort(struct CElem m[], unsigned l, unsigned r)
{ static unsigned incs[16] = { 1391376,463792,198768,86961,33936,
    13776,4592,1968,861,336,112,48,
    21,7,3,1 };
  unsigned i, j, k, h;
  struct CElem v;

  for (k = 0; k < 16; k++)
    for (h = incs[k], i = l+h; i <= r; i++) {
      v = m[i]; j = i;
      while (j > h && m[j-h].key > v.key) {
        m[j] = m[j-h];
        j -= h;
      }
      m[j] = v;
    }
}

```

Làm thế nào để chọn các bước chúng ta cần? Vấn đề này cực kỳ phức tạp và về mặt toán học vẫn chưa được giải quyết. Để có được lợi ích tối đa từ việc áp dụng nhất quán các phân loại, chúng cần phải tương tác ở mức tối đa. Điều này dẫn chúng ta đến ý tưởng rằng các bước nên là bội số của nhau, ví dụ như độ liên tiếp của cặp đôi.

Papernov và Stasevich chỉ ra rằng nếu trình tự được sử dụng

$$1, 3, 7, 15, 31, 63, 127, \dots, 2^k - 1, \dots$$

thuật toán yêu cầu $\Theta(n\sqrt{n})$ bước [Papernov] [Stasevic-1975].

Pratt cung cấp

$$1, 2, 3, 4, 6, 8, 9, 12, 16, \dots, 2^p 3^q, \dots,$$

điều này chứng tỏ rằng $\Theta(n(\log_2 n)^2)$ bước là cần thiết. Về mặt tiệm cận, đường của Pratt là tốt nhất, nhưng tiếc là nó phát triển chậm và quá dài (xem [Pratt-1979]). Knuth (xem [Knuth-3/1968]) cung cấp phạm vi

$$1, 4, 13, 40, 121, \dots,$$

được cho bởi các phương trình $\delta_{k-1} = 3\delta_k + 1, \delta_s = 1, s = [\log_3 n] - 1$, được viết theo thứ tự ngược lại. Một bộ truyện hay khác, cũng do Knut đề xuất, là:

$$1, 3, 7, 15, 31, \dots$$

được cho bởi các phương trình $\delta_{k-1} = 2\delta_k + 1, \delta_s = 1, s = [\log_2 n] - 1$, lại được viết theo thứ tự ngược lại. Trong trường hợp này, số phép so sánh có bậc là $n^{1/2}$. Kết quả tốt thu được đối với chuỗi đạt đến mức tối đa của việc giảm cấp số nhân của một tư nhân 1,7. Trong trường hợp này, độ phức tạp của thuật toán là $\Theta(n(\log_2 n)^2)$.

Bài tập

- ▷ 3.5. Chứng minh rằng nếu một dãy đã được sắp xếp, thì sau khi áp dụng δ' -sắp xếp, $\delta' < \delta$ nó tiếp tục được sắp xếp.

3.2.4. Phương pháp bong bóng

Phương pháp bong bóng chắc chắn là phổ biến nhất trong số các lập trình viên, đặc biệt là với số lượng phần tử sắp xếp đủ nhỏ. Như Bảng 3.2 dưới đây. tuy nhiên, sự phổ biến của nó hầu như không do tính hiệu quả của nó (Đây là thuật toán tệ nhất trong tất cả các thuật toán được thảo luận trong chương này!). "Sự quyền rũ" của ông có thể được tìm kiếm thay vì dễ dàng nó được hiện thực hóa, trong sự đơn giản trong ý tưởng của ông và có lẽ ở một mức độ nào đó trong cái tên kỳ lạ của ông. Thông thường, thuật toán sắp xếp đầu tiên mà một lập trình viên mới học về là phương pháp bong bóng. Không thể bỏ qua thực tế là phương pháp này là cơ sở để phân loại nhanh Hoor - thuật toán sắp xếp phổ quát nhanh nhất.

Ý tưởng của thuật toán là xem một cách nhất quán các phần tử của mảng và, nếu đổi với một cặp phần tử liền kề x_{i-1} và x_i thì kết quả là $x_{i-1} > x_i$, chúng sẽ hoán đổi vị trí cho nhau (giả sử sắp xếp theo thứ tự tăng dần). Trong quá trình sắp xếp, các phần tử nhẹ nhất, như bong bóng, nổi lên "bề mặt", tức là ở cuối bên trái của mảng. Do đó tên của phương thức.

Chương trình 3.6. Phương pháp bong bóng (306bubsort1.c)

```
void bubbleSort1(struct CElem m[], unsigned n)
{ unsigned i, j;
  for (i = 1; i < n; i++)
    for (j = n-1; j >= i; j--)
      if (m[j-1].key > m[j].key)
        swap(m+j-1, m+j);
}
```

Cách nổi bóng	Số tối thiểu	Số trung bình	Số tối đa
So sánh	$n(n - 1)/2$	$n(n - 1)/2$	$n(n - 1)/2$
Trao đổi	0	$3(n^2 - n)/4$	$3(n^2 - n)/2$

Bảng 3.2. So sánh và trao đổi trong sắp xếp theo cách nổi bong bóng.

Bảng 3.2 cung cấp thông tin về các đặc điểm chính của phân loại bong bóng. Có thể thấy, số phép so sánh được thực hiện bởi thuật toán luôn giống nhau, cụ thể là $n(n - 1)/2$. Điều này khiến chúng ta nghĩ rằng một số cải tiến là có thể. Ví dụ: chúng ta có thể thêm một cờ cho biết liệu một cuộc trao đổi đã diễn ra trên lần lặp hiện tại hay chưa. Quá trình phân loại sẽ tiếp tục miễn là các cuộc trao đổi diễn ra. Tuy nhiên, các thử nghiệm thực tế cho thấy rằng lợi ích của việc giới thiệu một cờ là gần như không thể nhận thấy và với một mảng được sắp xếp ngược lại, thuật toán thậm chí còn chậm lại.

Một cải tiến đáng kể khác có thể được thực hiện từ ý tưởng rằng việc xem xét lại các yếu tố chắc chắn đã biết là ở vị trí cuối cùng của chúng là vô nghĩa. Lần này chúng ta sẽ sắp xếp theo hướng ngược lại. Tất nhiên, phần sau là không đáng kể, nhưng nó mang lại một cái nhìn mới về thuật toán: chúng ta bắt đầu từ "dưới cùng" ở cuối mảng và các phần tử nặng nhất chìm dần về phía nó. Chúng ta sẽ

duy trì biến i , cho biết chỉ số tối đa mà lần lặp trước đó đã được trao đổi. Rõ ràng, các yếu tố bên phải của cuộc trao đổi cuối cùng nằm ở vị trí cuối cùng của chúng, điều này cho chúng ta lý do để không nhìn vào chúng nữa.

Chương trình 3.7. Phương pháp bong bóng (307bubsort2.c)

```
void bubbleSort2(struct CElem m[], unsigned n)
{ unsigned i, j, k;
  for (i = n; i > 0; i = k)
    for (k = j = 0; j < i; j++)
      if (m[j].key > m[j+1].key) {
        swap(m+j, m+j+1);
        k = j;
      }
}
```

Bài tập

- ▷ 3.6. Tìm công thức trong Bảng 3.2

3.2.5. Sắp xếp bằng cách lắc

Mặc dù hiệu quả thấp, phương pháp bong bóng cho phép một số cải tiến. Chúng ta đã đề cập đến hai trong số chúng ở trên - thêm một cờ và duy trì một biến cho biết chỉ số trao đổi tối đa từ bước trước. Có một tính năng khác: một phần tử sáng riêng biệt ở cuối mảng sẽ xuất hiện trong một lần lặp duy nhất, trong khi phần tử nặng "đơn độc" nằm gần bề mặt sẽ chìm ở mỗi bước của thuật toán chỉ một mức xuống dưới cùng. Nếu chúng ta đảo ngược hướng của các chu kỳ, mọi thứ sẽ hoàn toàn ngược lại, nhưng sự bất đối xứng rõ ràng sẽ vẫn còn. Điều này khiến chúng ta nghĩ đến việc đảo ngược hướng sắp xếp ở mỗi bước. Vì vậy, chúng ta nhận được cải tiến thứ ba của thuật toán. Áp dụng ba cải tiến cùng lúc, chúng ta nhận được một thuật toán mới - sắp xếp bằng cách lắc:

Chương trình 3.8. Sắp xếp bằng cách lắc (308shaker.c)

```
void shakerSort(struct CElem m[], unsigned n)
{ unsigned k = n, r = n-1;
  unsigned l = 1, j;
```

```

do {
    for (j = r; j >= 1; j--)
        if (m[j-1].key > m[j].key) {
            swap(m+j-1,m+j);
            k = j;
        }
    l = k + 1;
    for (j = l; j <= r; j++)
        if (m[j-1].key > m[j].key) {
            swap(m+j-1,m+j);
            k = j;
        }
    r = k - 1;
} while (l <= r);
}

```

Trong trường hợp tốt nhất, thuật toán trên thực hiện $n - 1$ so sánh và trong trường hợp giữa - lại $\Theta(n^2)$. Mặc dù có một số cải thiện về mặt này so với phương pháp bong bóng cổ điển, nhưng hiệu quả tổng thể của thuật toán không cải thiện nhiều, vì sắp xếp theo kiểu rung chuyển không làm giảm số lượng trao đổi được thực hiện. Và về nguyên tắc, cuộc trao đổi khó hơn nhiều lần so với việc so sánh.

Bài tập

▷ 3.7. So sánh các tùy chọn sắp xếp bong bóng khác nhau:

- a) bong bóng không có cờ;
- b) bong bóng có cờ;
- c) phân loại đá.

3.2.6. Nhanh chóng phân loại Hoor

Các thuật toán dựa trên trao đổi phần tử được thảo luận ở trên có hiệu quả khá khiêm tốn, vì phương pháp bong bóng cổ điển thậm chí có các tham số tồi tệ nhất so với tất cả các thuật toán sắp xếp được thảo luận cho đến nay. Mặc dù chúng ta đã nỗ lực rất nhiều (ba cải tiến đã được thực hiện trong quá trình phân loại!), Những đặc điểm này không cải thiện nhiều. Như thế phân loại thông qua

trao đổi không dẫn đến các thuật toán hiệu quả ... Có thực sự là như vậy không?

Trong quá trình rung chuyển, mặc dù đã có những cải tiến, nhưng chúng ta không đạt được kết quả khả quan, bởi vì, giảm số lượng so sánh, chúng ta không ảnh hưởng đến số lượng trao đổi của các phần tử. Nay giờ chúng ta sẽ quay lại: chúng ta sẽ cố gắng chủ yếu để giảm thiểu số lượng trao đổi. Vì việc trao đổi hai yếu tố đắt hơn nhiều lần so với việc so sánh, nên một sự cải thiện nghiêm trọng có thể được mong đợi. Nhưng làm thế nào để giảm số lượng trao đổi? Chúng ta hãy nhớ lại cải tiến thứ ba của phương pháp bong bóng (xem 3.2.4), Mà chúng ta đã thực hiện trong quá trình thực hiện lắc (xem 3.2.5): Đảo ngược hướng truyền của các phần tử ở mỗi bước của thuật toán. Lý do cho điều này là do chúng ta quan sát thấy rằng một nguyên tố nặng duy nhất nằm gần bề mặt sẽ chìm ở mỗi bậc chỉ với một vị trí ở phía dưới. Đồng thời, một phần tử ánh sáng riêng biệt nằm ở cuối mảng sẽ xuất hiện đến vị trí cuối cùng của nó trong một lần lặp lại duy nhất. Tuy nhiên, lưu ý rằng ngay cả trong trường hợp này, phần tử ánh sáng sẽ phải được trao đổi với tất cả các phần tử trên đường đến vị trí cuối cùng của nó, và thao tác này rất tốn kém! Làm thế nào để giảm số lượng trao đổi? Trong trường hợp mảng nghịch đảo, chúng ta có thể hoán đổi phần tử đầu tiên với phần tử cuối cùng, phần tử thứ hai với phần tử áp chót, v.v., sắp xếp toàn bộ mảng cho $[n/2]$ các hoán đổi. Kết luận là hiển nhiên: khoảng cách giữa các sàn giao dịch càng lớn thì hiệu quả của chúng càng cao.

Ý tưởng của thuật toán do Hoor đề xuất là chọn một phần tử x và chia mảng thành hai phân vùng: bên trái, trong đó các phần tử nhỏ hơn x và bên phải, trong đó chúng lớn hơn. Chúng ta áp dụng cùng một thuật toán cho các phần bên trái và bên phải, giảm dần ranh giới bên trái và bên phải của các mảng con được xem xét cho đến khi chúng ta đạt đến các khoảng chứa một phần tử duy nhất. Sau khi thuật toán hoàn thành, mảng sẽ được sắp xếp. (Tại sao?)

Gọi q là chỉ số của x trong mảng, tức là $x = m[q]$. Hãy sắp xếp mảng con $m[l, l + 1, \dots, r]$ và partition () chia nó thành hai phần: bên trái ($m[l, l + 1, \dots, q]$) và bên phải ($m[q + 1, q + 2, \dots, r]$) và trả về kết quả là q . Lưu ý rằng sự tách biệt gần như chắc chắn liên quan đến

sự trao đổi của các phần tử, tức là partition() không tìm kiếm một phần tử x trong mảng, nhưng chọn nó và chia nó thành hai vùng tùy theo giá trị của nó. Dưới đây chúng ta sẽ xem cách này có thể được thực hiện.

Sắp xếp nhanh có thể được viết theo cách này (Mảng m[]) không được truyền dưới dạng tham số, nhưng được coi là một biến toàn cục: điều này tiết kiệm không gian ngăn xếp và tăng tốc độ vì hàm là đệ quy.):

```
void quickSort(int l, int r)
{ int q;
  if (l < r) {
    q = partition(l,r);
    quickSort(l,q);
    quickSort(q+1,r);
  }
}
```

Giả sử rằng dựa trên một số công thức, chúng ta đã tính được q . (Đầu tiên chúng ta sẽ chọn phần tử ngoài cùng bên trái hoặc ngoài cùng bên phải của mảng trước phân vùng.) Lưu ý rằng $m[q]$ chứa giá trị của x trước phân vùng. Sau phép chia q sẽ là một chỉ số biên: ở bên trái của nó (bao gồm) chúng ta sẽ có các phần tử có khóa không vượt quá giá trị của x và ở bên phải - khóa lớn hơn x .

Câu hỏi quan trọng tiếp theo được đặt ra: Làm thế nào để thực hiện việc phân chia? Nhìn chung có hai cách tiếp cận khác nhau. Đầu tiên sau đây:

```
unsigned partition(int l, int r)
{ int q, j, x;
  q = l - 1; x = m[r].key;
  for (j = l; j <= r; j++)
    if (m[j].key <= x) {
      q++;
      swap(m+q,m+j);
    }
  if (q == r) q--;
  return q;
}
```

Phương pháp đề xuất hoạt động như thế nào? Chọn phần tử x để thực hiện tách. Các phần tử nhỏ hơn hoặc bằng x phải nằm ở bên trái của mảng và lớn hơn x phải ở bên phải. Phần được coi là của mảng $m[l, l + 1, \dots, r]$ được xem từ trái sang phải, trong khi phần bên trái của nó là một vùng không ngừng mở rộng của các phần tử nhỏ hơn hoặc bằng x được xây dựng. Cuối bên phải của khu vực được xác định bởi q . Khi j đến cuối r của phân vùng, q sẽ chỉ ra ranh giới giữa hai vùng. (Lưu ý: Trong một chương trình thực, bạn nên thay thế hàm `swap()` bằng mã của nó để đạt hiệu quả cao hơn.)

Một phương pháp khả thi khác là sử dụng hai chỉ số i và j , cho thấy ranh giới của hai khu vực liên tục mở rộng từ hai đầu đến trung tâm. Ở mỗi bước, một nỗ lực được thực hiện để mở rộng vùng bên trái sang bên phải càng lâu càng tốt, nghĩa là miễn là có một phần tử nhỏ hơn hoặc bằng x ở bên phải của nó. Điều tương tự cũng được thực hiện đối với khu vực bên phải (cả hai chu kỳ trong khi). Sau đó, vị trí của hai phần tử kết giới được hoán đổi cho các khu vực bên trái và bên phải đã ngừng mở rộng, và quá trình này được lặp lại từ đầu. Kết thúc ở "cuộc gặp gỡ" của hai lĩnh vực. Chỉ số i cho biết điểm cuối bên phải của vùng bên trái và j - điểm cuối bên trái của vùng bên phải:

```
unsigned partition(int l, int r)
{ unsigned i, j, x;
  i = l; j = r; x = m[l].key;
  do {
    while (x > m[i].key) i++;
    while (x < m[j].key) j--;
    if (i <= j) {
      swap(m+i,m+j);
      i++; j--;
    }
  while (i <= j);
  return j;
}
```

Sau khi kết hợp `partition()` và `quickSort()`, chúng ta nhận được (cho tùy chọn đầu tiên):

```

void quickSort(int l, int r)
{ int i, j, x;
  i = l-1; x = m[r].key;
  for (j = 1; j <= r; j++)
    if (m[j].key <= x) {
      i++;
      swap(m+i,m+j);
    }
    if (i == r) /* Tất cả các phần tử đều <= x. Diện tích giảm đi 1. */
      i--;
    if (l < i)
      quickSort(l, i);
    if ((i+1) < r) /**
      quickSort(i+1,r); /**
}

```

Đối với phương án thứ hai, chúng ta có:

Chương trình 3.9. Sắp xếp nhanh (309qsort.c)

```

void quickSort(int l, int r)
{ int i, j, x;
  i = l;
  j = r;
  x = m[r].key;
  do {
    while (x > m[i].key) i++;
    while (x < m[j].key) j--;
    if (i <= j) {
      swap(m+i, m+j);
      i++;
      j--;
    }
  } while (j >= i);
  if (j > 1)
    quickSort(l, j);
  if (i < r) /**
    quickSort(i, r); /**
}

```

Hai phép đảo ngược đê quy liên tiếp được thực hiện để sắp xếp

hai phần của mảng. Rõ ràng là chúng không thể được thực hiện cùng một lúc và việc sau này luôn bị hoãn lại. Điều này hóa ra là "không cần thiết" theo nghĩa là không cần đệ quy mới - bây giờ chúng ta có thể xử lý lặp đi lặp lại. Một số trình biên dịch có thể tự động nhận dạng và loại bỏ đệ quy thứ hai. Tất nhiên, với nỗ lực tối thiểu, chúng ta có thể tự mình làm điều đó, nhận được một phiên bản bán lặp lại. Vì mục đích này, chỉ cần thay thế các hàng được đánh dấu bằng $/ \text{***} /$ bằng phép gán $1 = i + 1;$ (đối với biến thể đầu tiên) và $1 = i;$ (đối với tùy chọn thứ hai), sau đó lặp lại các hành động của hàm một lần nữa. Với mục đích này, chúng ta sẽ phải "mặc quần áo" cho cô ấy trong một chu kỳ khác.

Nếu chúng ta muốn, chúng ta có thể loại bỏ hoàn toàn đệ quy. Với mục đích này, chúng ta sẽ sử dụng cách tiêu chuẩn để loại bỏ nó - sử dụng ngăn xếp. Chúng ta sẽ lưu ý rằng đệ quy và lặp lại trong quá trình phát triển các thuật toán là hoàn toàn có thể hoán đổi cho nhau. Thực vậy, mỗi chương trình có thể được mô phỏng bằng đệ quy. Ở chiều ngược lại, mọi thứ phức tạp hơn. Chúng ta sẽ chỉ lưu ý rằng máy tính là máy lặp, và do đó mọi đệ quy chúng ta viết cuối cùng đều chuyển sang lặp. Mặt khác, đệ quy được máy tính mô hình hóa bằng cách đặt một số dữ liệu nhất định vào ngăn xếp hệ thống. Vậy tại sao không sử dụng ngăn xếp của riêng chúng ta một cách rõ ràng? Với mục đích này, ở mỗi bước, phần bên trái của mảng sẽ được xử lý trực tiếp, trong khi phần bên phải sẽ được đặt trong ngăn xếp và quá trình xử lý của nó sẽ bị hoãn lại cho đến phần cuối cùng với phần bên trái và tất cả các bài toán con có liên quan (ở đây x chúng ta chọn làm phần tử ở giữa mảng con được xem xét):

```
void quickSort(void)
{ int i, j, l, r, s, x;
struct { int l, r; } stack[MAX];
stack[s = 0].l = 0;
stack[0].r = n-1;
for (;;) {
    l = stack[s].l;
    r = stack[s].r;
    if (0 == s--)
        break;
    do {
```

```

i = l; j = r; x = m[(l+r)/2].key;
do {
    while (m[i].key < x) i++;
    while (m[j].key > x) j--;
    if (i <= j) {
        swap(m+i,m+j);
        i++;
        j--;
    }
} while (i<=j);
if (i < r) { /**
    stack[++s].l = i; /**
    stack[s].r := r; /**
} /**
r = j; /**
} while (l < r);
}
}

```

Chúng ta hãy thử đánh giá ngăn xếp phần mềm cần thiết trong hoạt động của giải pháp lặp lại được đề xuất. Rõ ràng là trong trường hợp tối ưu, chúng ta sẽ giảm một nửa phân vùng mỗi lần, dẫn đến nghịch lưu đệ quy $\Theta(\log_2 n)$ ngăn xếp. Tuy nhiên, trong trường hợp xấu nhất, với sự lựa chọn sai phần tử, kích thước của phân vùng được đề cập sẽ chỉ giảm 1 lần mỗi lần, nơi chúng ta sẽ có $\Theta(n)$ ngăn xếp. Trong phiên bản đệ quy, mọi thứ thậm chí còn tồi tệ hơn, bởi vì có nhiều dữ liệu hơn trong ngăn xếp: địa chỉ trả về, các tham số và các biến cục bộ. Một cách có thể giải quyết là xếp chồng các phần dài hơn và xem xét các phần ngắn hơn ngay lập tức. Không khó để thấy rằng trong cách tiếp cận này, ngăn xếp được yêu cầu sẽ luôn là $\Theta(\log_2 n)$. Vì mục đích này, các dòng được đánh dấu bằng `/ *** /` nên được thay thế bằng đoạn chương trình sau (Tại sao?):

```

/* Đặt phần lớn hơn vào ngăn xếp*/
if (j - 1 < r - i) {
    /* Chèn phần bên phải vào ngăn xếp*/
    if (i < r) {
        stack[++s].l = i;
        stack[s].r = r;
    }
}

```

```

    r = j;
}
else {
    /* Chèn phần bên trái vào ngăn xếp */
    if (l < j)
        stack[++s].l = l;
        stack[s].r = j;
    }
    l = i;
}

```

Rõ ràng rằng các phần tử ở bên trái và bên phải của mảng càng được phân bố đều do kết quả của việc phân tách, thì càng cần ít bước hơn. Trong trường hợp tốt nhất, nếu chúng ta chọn phần tử cỡ vừa mỗi lần, $\log_2 n$ phần vùng sẽ đủ cho việc sắp xếp tổng thể của mảng. Với tổng số phép so sánh trong trường hợp tối ưu, chúng ta nhận được $n \cdot \log_2 n$. Số lượng trao đổi dự kiến tại mỗi lần tách có thứ tự là $n/6$. Do đó số lần trao đổi trong trường hợp tối ưu là $(n/6) \cdot \log_2 n$. Mặc dù xác suất bắn trúng phần tử ở giữa là $1/n$, hiệu quả trung bình của việc sắp xếp nhanh không phụ thuộc vào n , và nó kém hơn mức trung bình chỉ $2 \cdot \ln 2$ [Knuth-3/1968].

Quay trở lại câu hỏi: Làm thế nào để chọn phần tử x , mà chúng ta chia mảng? Một cách tiếp cận khả thi là luôn lấy một phần tử cố định từ mảng, chẳng hạn như phần tử đầu tiên, cuối cùng hoặc giữa (như chúng ta đã làm ở trên). Việc lựa chọn một phần tử phân chia là cực kỳ quan trọng đối với tốc độ của thuật toán: khóa của phần tử được chọn càng gần khóa của phần tử cỡ vừa cho phân vùng, thì việc chia mảng thành hai phần càng đồng đều và do đó độ sâu của cây đệ quy càng nhỏ, tức là thuật toán càng hiệu quả. (Tại sao?)

Do đó, trung vị của hai hoặc nhiều phần tử đôi khi được chọn làm phần tử phân chia. Tuy nhiên, điều này có liên quan đến các hoạt động bổ sung, và do đó dẫn đến sự chậm trễ trong lựa chọn, điều này không phải lúc nào cũng được bù đắp bằng lợi ích thu được và không nhất thiết dẫn đến cải tiến. Chúng ta khuyên người đọc quay lại *sắp xếp nhanh* sau, sau khi đã làm quen với phiên bản đệ quy của *sắp xếp bit* từ 3.3.3, Và để so sánh cách chọn phần tử phân tách.

Phân loại nhanh ẩn chứa một số điều bất ngờ. Như đã đề cập ở trên, đây là thuật toán sắp xếp phổ quát nhanh nhất được biết đến (trong 3.2. Chúng ta sẽ thấy rằng trong một số trường hợp đặc biệt quan trọng với các hạn chế bổ sung thì có các thuật toán tuyển tính!). Độ phức tạp thuật toán trung bình của nó là $\Theta(n \log_2 n)$, cũng như độ phức tạp của đối thủ cạnh tranh chính - sắp xếp theo hình chóp (xem 3.2.9), sẽ được thảo luận sau. Tuy nhiên, thực tế cho thấy rằng phân loại nhanh sẽ "đánh bại" mức trung bình của hình chóp từ hai đến ba lần. Đồng thời, phân loại của Hoar thất thường hơn nhiều và nếu chọn sai x , hiệu quả của nó có thể giảm đáng kể. Trong trường hợp xấu nhất, khi mỗi lần kích thước của phần được đề cập chỉ giảm đi 1, thì độ phức tạp của nó tỷ lệ với n^2 . [Knuth-3/1968] [Wirth-1980]

Điểm mạnh của phân loại nhanh là ở cờ bạc của nó. Mảng càng xáo trộn, hiệu quả của phương pháp càng lớn. Tác giả Hoar khuyến nghị rằng việc lựa chọn x được thực hiện một cách ngẫu nhiên hoặc thậm chí là giá trị trung bình cộng của ba hoặc nhiều phần tử được chọn ngẫu nhiên. Trong thực tế, một chiến lược như vậy hầu như không ảnh hưởng đến thuật toán nói chung, trong khi tăng đáng kể hiệu quả của nó trong trường hợp xấu nhất. Cần lưu ý rằng, giống như hầu hết các phương pháp nhanh hiện đại, sức mạnh của sắp xếp nhanh không được biểu hiện ở một số lượng nhỏ các phần tử. Một cách tiếp cận tốt để loại bỏ thiểu số này là áp dụng sắp xếp nhanh cho các phân vùng có ít nhất k (ví dụ: $k = 20$) phần tử, sử dụng thuật toán bong bóng đơn giản hơn cho các phân vùng nhỏ hơn (xem 3.2.4), lựa chọn trực tiếp (xem 3.2.8) hoặc chèn (xem 3.2.2). Một chiến lược thậm chí còn phức tạp hơn là không làm gì khi đến một phân vùng có ít hơn k phần tử. Sau khi hoàn thành việc sắp xếp nhanh đã sửa đổi, mảng không được sắp xếp theo thứ tự mà được chia thành nhiều nhóm nhỏ gồm các phần tử có giá trị tương tự nhau và mỗi phần tử của nhóm bất kỳ đều lớn hơn các phần tử của tất cả các nhóm ở bên trái được xem xét. Bằng cách này, mảng gần như được sắp xếp và là một đầu vào tốt để sắp xếp bằng cách chèn: các phần chèn sẽ được thực hiện trong khu vực tương ứng, nghĩa là được đảm bảo ở khoảng cách ngắn.

Bài tập

- ▷ 3.8. Chứng minh rằng phân loại nhanh Hoor sắp xếp từng trình tự đầu vào một cách chính xác.
- ▷ 3.9. Chứng minh rằng sắp xếp Hoor nhanh yêu cầu một chồng có thứ tự $\Theta(\log_2 n)$.
- ▷ 3.10. Để kiểm tra việc triển khai phần mềm được đề xuất của phân loại nhanh Hoor cho tất cả các trình tự đầu vào có thể có với tối đa 32 phần tử:
- cho một dãy số ngẫu nhiên;
 - với mọi hoán vị của các số từ 1 đến n ;
 - cho tất cả các hoán vị của các số trong một tập hợp nhiều (với các phần tử lặp lại);
 - sử dụng nguyên tắc số không và số một.
- ▷ 3.11. Làm các bài tập kiểm tra thực nghiệm cho từng tiêu mục của bài toán trên và so sánh kết quả với lý thuyết.
- ▷ 3.12. Tìm ra một biến thể lặp đi lặp lại của sắp xếp nhanh Hoor.
- ▷ 3.13. Số phần tử trong phân vùng đã sắp xếp nên sử dụng thuật toán đơn giản như sắp xếp chèn để sắp xếp nhanh?
- ▷ 3.14. Chứng minh rằng phần tử càng gần trung vị là phần tử mà sự phân chia hai phần xảy ra trong sắp xếp nhanh Hoor, thuật toán càng hiệu quả.
- ▷ 3.15. Để xác định về mặt lý thuyết, cách tốt nhất để chọn một phần tử phân tách trong phân loại nhanh Hoor (xem 3.2.6):
- đầu tiên;
 - cuối cùng;
 - ở giữa;
 - trung bình cộng của hai phần tử;
 - trung bình cộng của ba phần tử;
 - trung vị: phần tử cõi trung bình trên mỗi cõi phiếu (xem ??);
 - một phần tử nhân tạo, ví dụ như trong phân loại bit (xem 3.3.3).

- ▷ 3.16. Thực hiện các bài kiểm tra thực nghiệm cho từng tiêu mục của bài toán trên và so sánh kết quả với lý thuyết.
- ▷ 3.17. Đề xuất sửa đổi cách sắp xếp Hoor nhanh, điều này sẽ đảm bảo độ phức tạp về thời gian $\Theta(n \log_2 n)$ trong trường hợp xấu nhất.
- ▷ 3.18. Để thay đổi kiểu tham số chính thức của hàm quickSort() từ **int** sang **unsigned**. Sự cố nào xảy ra và tại sao? Bạn đưa ra giải pháp nào?
- ▷ 3.19. Đối với mỗi n tự nhiên để tìm một chuỗi đầu vào trong đó sắp xếp nhanh có độ phức tạp:
 - $\Theta(n \log_2 n)$
 - $\Theta(n^2)$

3.2.7. Phương pháp Thỏ và Rùa

Chúng ta sẽ cố gắng cải thiện hiệu quả khiêm tốn của phương pháp bong bóng (xem 3.2.4) Bằng cách đi theo hướng khác. Như chúng ta đã nhiều lần lưu ý ở trên (xem 3.2.5 và 3.2.6), nhược điểm chính của thuật toán là tính không đối xứng trong hành vi của nó - một phần tử nhẹ từ cuối mảng "xuất hiện" ngay lập tức, trong khi một phần tử nặng duy nhất ở mỗi bước "chìm" xuống một mức duy nhất. Các yếu tố ánh sáng, tức là những yếu tố di chuyển nhanh, sẽ được gọi là thỏ, và những yếu tố chậm - rùa. Các quan sát cho thấy hầu hết mọi chuỗi yếu tố lớn đều chứa một con rùa, và mỗi con rùa dẫn đến độ trễ tối đa.

Làm thế nào để giảm thiệt hại từ rùa? Sự thay đổi về hướng thu thập thông tin mà chúng ta áp dụng cho thuật toán sắp xếp bập bênh (xem 3.2.5) chỉ đơn giản là hoán đổi vai trò của thỏ và rùa. Năm 1991, trong hai bài báo liên tiếp trên tạp chí Byte, Lacey và Box đề xuất một cách tiếp cận triệt để khác - loại bỏ rùa bằng cách cho phép chúng "nhảy" đến vị trí cuối cùng thay vì "bò". Vì mục đích này, so sánh được thực hiện giữa các phần tử ở xa thay vì các phần tử liền kề, như trong bong bóng cổ điển. Khoảng cách giữa các phần tử được so sánh được xác định theo bước giảm dần.

Làm thế nào để chọn bước? Sau khoảng 200.000 lần thử, Lacey và Box đã đi đến kết luận thực nghiệm rằng bước giảm tối ưu (hệ số

co) là 1,3. Khi bạn đến bước 1, thuật toán đã loại bỏ các con rùa và hoạt động như một bong bóng bình thường. Ở các giá trị bước nhỏ hơn 1,3, thuật toán chậm lại do số lượng phép so sánh thừa tăng lên và ở các giá trị cao hơn do va phải một vài con rùa. Hai người cũng đã thực nghiệm phát hiện ra rằng việc phân chia mảng thành các khu vực, mỗi khu vực được xử lý bằng một bước khác nhau, cũng như việc sử dụng hệ số co ngót giảm dần hoặc thay đổi theo cấp số nhân, không dẫn đến cải thiện.

Tuy nhiên, họ đã cố gắng thực hiện một số cải tiến, mà họ xây dựng thành "Quy tắc 11". Ý tưởng là sử dụng 11 thay vì các bước của 9 và 10. Với hệ số co rút gần với mức tối ưu 1,3, bước này có thể được giảm xuống 1 trong ba cách sau:

```
9 6 4 3 2 1
10 7 5 3 2 1
11 8 6 4 3 2 1
```

Theo cách thứ ba, các con rùa nhỏ biến mất trước khi bước trở thành 1. Đồng thời, trong hai cách đầu tiên, với xác suất 8%, danh sách vẫn chứa các con rùa nhỏ, do đó thuật toán làm chậm đi 15 -20%.

Thuật toán được mô tả theo cách này rất giống với sắp xếp Shell (xem 3.2.3) - cả hai đều sử dụng hệ số thu nhỏ. Tuy nhiên, không khó để tin rằng sự giống nhau chỉ là bên ngoài. Thuật toán của Shell là một phép chèn (xem 3.2.2) Với bước giảm dần, trong khi phương pháp của thỏ và rùa dựa trên ý tưởng bong bóng (xem 3.2.4). Ngoài ra, Shell thực hiện sắp xếp hoàn chỉnh các danh sách con có liên quan ở mỗi bước.

Do đó một số khác biệt khác. Trong khi đối với Shell hệ số co rút tối ưu là 1,7, đối với "thỏ và rùa" là 1,3. Cuối cùng, độ phức tạp thuật toán của Shell là $\Theta(n \cdot (\log_2 n)^2)$, trong khi ở thỏ và rùa là $\Theta(n \cdot \log_2 n)$ trong cả trường hợp trung bình và xấu nhất. Điều này xếp hạng phương pháp của Lacey và Boxing trong số các phương pháp phân loại hiện đại nhanh nhất. Sau đây là một ví dụ triển khai:

Chương trình 3.10. Thuật toán thỏ và rùa (310combsort.c)

```
void combSort(struct CElem m[], unsigned n)
{ unsigned s, i, j, gap = n;
  do {
```

```

s = 0;
gap = (unsigned) (gap/1.3);
if (gap < 1)
    gap = 1;
for (i = 0; i < n-gap; i++) {
    j = i + gap;
    if (m[i].key > m[j].key) {
        swap(m+i, m+j);
        s++;
    }
}
} while (s != 0 || gap > 1);
}

```

Như một nhược điểm nhất định của việc phân loại theo phương pháp thỏ và rùa, chúng ta có thể chỉ ra công việc với số đầu phẩy động và sử dụng phép toán chia nặng. Thay vào đó, có thể dễ dàng tránh được phép chia bằng cách sử dụng phép nhân. Với mục đích này, đơn đặt hàng:

gap = (long) (gap / 1,3);
có thể được thay thế bởi:
gap = (long) (gap * 0,76923076923);

Quá trình chuyển đổi sang các phép toán số nguyên được thực hiện với:

gap = gap * 8/11;

Chúng ta có thể loại bỏ sự phân chia bằng cách di chuyển xa hơn ra khỏi phạm vi tối ưu, nhưng tăng tốc đáng kể các tính toán:

gap = gap * 6 >> 3.

Kể từ khi bài báo được xuất bản, một số nỗ lực đã được thực hiện để cải thiện nó, trong đó quan trọng nhất là của Jim Will. Nó gợi ý sử dụng hằng số 1,279604943109628 thay vì 1,3. Trên thực tế, anh ta không sử dụng nó trực tiếp, nhưng đề nghị rằng các bước được cố định trước. Sau đây là trình tự của nó (được sử dụng theo thứ tự giảm dần) cho các mảng có tối đa 4 tỷ phần tử:

11, 13, 17, 23, 29, 37, 47, 61, 79, 103, 131, 167, 216, 277, 353, 449
, 577, 739, 947, 1213, 1553, 1987, 2543, 3259, 4166, 5333, 6829,

8741, 11177, 14310, 18313, 23431, 29989, 38371, 49103, 62827, 80
 407, 102881, 131648, 168463, 215573, 275840, 352973, 451669, 5779
 57, 739560, 946346, 1210949, 1549547 253720, 2147483647

Bài tập

- ▷ 3.20. Chứng minh rằng phương pháp của thỏ và rùa có độ phức tạp $\Theta(n \log_2 n)$ cả ở giữa và trong trường hợp xấu nhất.
- ▷ 3.21. Để so sánh về mặt lý thuyết và thực nghiệm thuật toán Shell (xem 3.2.3) Và phương pháp của thỏ và rùa.

3.2.8. Sắp xếp lựa chọn trực tiếp

Một phương pháp sắp xếp cơ bản khác có độ phức tạp $\Theta(n^2)$ là phương pháp chọn trực tiếp hay còn gọi là phương pháp chọn trực tiếp. Mảng được chia thành phần được sắp xếp và phần chưa được sắp xếp, và ở mỗi bước của thuật toán, vùng được sắp xếp sẽ được mở rộng sang bên phải một phần tử. Bước đầu tiên là phần tử tối thiểu của mảng và được trao đổi với phần đầu tiên. Trong bước thứ hai, phần tử tối thiểu khác được trao đổi với phần tử thứ hai của mảng, v.v. Ở mỗi bước tiếp theo, phần tử nhỏ nhất của phần chưa được sắp xếp sẽ hoán đổi với phần tử đầu tiên của phần chưa được sắp xếp (lưu ý rằng nó lớn hơn hoặc bằng mỗi phần tử của phần chưa được sắp xếp, tại sao?). Do đó mở rộng phần được sắp xếp trong khi nó không bao gồm toàn bộ mảng.

Chương trình 3.11. Sắp xếp lựa chọn trực tiếp (311selsort.c)

```
void straightSelection (struct CElem m[], unsigned n)
{ unsigned i, j;
  for (i = 0; i < n-1; i++)
    for (j = i+1; j <= n; j++)
      if (m[i].key > m[j].key)
        swap(m+i, m+j);
}
```

Một phiên bản sửa đổi một chút của việc triển khai ở trên cũng phổ biến: Trong chương trình tiếp theo, sự khác biệt là trong vòng lặp bên trong, địa chỉ của phần tử thứ j là $m[j]$ được lưu và vì mục đích này, biến x là đã giới thiệu.

Chương trình 3.12. Sắp xếp lựa chọn trực tiếp (312selsort2.c)

```

void straightSelection (struct CElem m[], unsigned n)
{
    unsigned i, j, ind;
    struct CElem x;
    for (i = 0; i < n - 1; i++)
        for (x = m[ind = i], j = i + 1; j < n; j++)
            if (m[j].key < x.key) {
                x = m[ind = j];
                m[ind] = m[i];
                m[i] = x;
            }
}

```

Chọn trực tiếp	Số tối thiểu	Số trung bình	Số tối đa
So sánh	$n(n - 1)/2$	$n(n - 1)/2$	$n(n - 1)/2$
Trao đổi	$3(n - 1)$	$n \ln n$	$[n^2/4] + 3(n - 1)$

Bảng 3.3. So sánh và trao đổi trong sắp xếp theo cách chọn trực tiếp.

Tương tự như sắp xếp bong bóng, thuật toán lựa chọn trực tiếp luôn thực hiện $n(n - 1)/2$ phép so sánh, như trong Bảng 3.3. Nói chung, thuật toán lựa chọn trực tiếp thích hợp hơn phương pháp bong bóng, mặc dù trong trường hợp đặc biệt của một mảng được sắp xếp trước hoặc sắp xếp gần hết, bong bóng cho kết quả tốt hơn.

Bài tập

▷ 3.22. Chứng minh rằng trong lựa chọn trực tiếp sắp xếp, mỗi phần tử của phần không được sắp xếp lớn hơn hoặc bằng mỗi phần tử của phần đã sắp xếp.

▷ 3.23. Để so sánh về mặt lý thuyết và thực nghiệm hai phương án sắp xếp lựa chọn trực tiếp.

▷ 3.24. Trích xuất các công thức từ Bảng 3.3.

3.2.9. Sắp xếp kim tự tháp Williams

Sắp xếp theo lựa chọn trực tiếp (xem 3.2.8) Xếp hạng trong số những cách kém hiệu quả nhất, chẳng hạn như sắp xếp theo bong

bóng (xem 3.2.4) Và chèn trực tiếp (xem 3.2.2). Hạn chế chính của nó là ở bước thứ i, luôn luôn thực hiện chính xác n-i so sánh, bất kể dữ liệu đầu vào là gì.

Trên thực tế, thuật toán tìm phần tử nhỏ nhất của mảng n phần tử yêu cầu chính xác $n - 1$ phép so sánh. Thật vậy, trong mỗi phép so sánh có đúng một ứng cử viên bị loại, và để xác định yếu tố tối thiểu, cần loại bỏ đúng $n - 1$ ứng viên.

Do đó, trong bước đầu tiên của một thuật toán tùy ý dựa trên sự lựa chọn, $n - 1$ phép so sánh sẽ được thực hiện và kết quả này, theo các cân nhắc ở trên, không thể cải thiện được. Thoạt nhìn, có vẻ như bước thứ hai sẽ yêu cầu thêm $n - 2$ phép so sánh. Suy nghĩ một cách linh hoạt, chúng ta nhận thấy rằng ở bước thứ i, chúng ta sẽ cần chính xác $n - i$ so sánh. Chúng ta không thể cải thiện đánh giá này? Người đọc chú ý chắc hẳn đã nhận thấy rằng bước đầu tiên và bước thứ hai của thuật toán không hoàn toàn bằng nhau. Thật vậy, kết quả của việc áp dụng bước đầu tiên, chúng ta không chỉ thu được phần tử tối thiểu của mảng mà còn có một số mối quan hệ giữa các cặp giữa các phần tử khác mà chúng ta có thể lưu trữ và sử dụng trong bước thứ hai. Rõ ràng là chúng ta càng nhận được nhiều tỷ lệ bổ sung, thì chúng ta sẽ cần ít so sánh hơn trong các bước tiếp theo, tức là cây so sánh nhị phân càng thấp và càng nhiều nhánh (xem 3.2.1) thì càng mang nhiều thông tin hơn.

Có thể thu được một cây so sánh tốt bằng cách sử dụng cơ chế giải đấu loại trừ (xem [Knuth-3/1968], [Reingold, Nivergelt, Deo-1980]), cơ chế này xây dựng cây so sánh từ lá đến gốc. Trong vòng đầu tiên của giải đấu, các trận đấu $x_1 : x_2, x_3 : x_4, \dots, x_{n-1} : x_n$ được diễn ra. Ở vòng thứ hai, các cặp thắng ở vòng trước sẽ đấu, v.v. Trong trận chung kết, hai người tham gia gặp nhau và xác định nhà vô địch của giải đấu. Nếu trong vòng thứ i của giải đấu mà số người tham gia là số lẻ (và điều này sẽ xảy ra ít nhất một lần, nếu n không phải là bậc 2, tại sao?), thì một trong những người chơi chết, tự động đủ điều kiện cho người tiếp theo vòng. Sử dụng chiến lược đã chỉ ra để xác định phần tử tối thiểu của một mảng, chúng ta nhận được: Trong vòng đầu tiên của giải đấu với $n/2$ phép so sánh, phần tử tối thiểu cho mỗi cặp được xác định và với $n/4$ phép so sánh, phần tử tối thiểu giữa hai cặp là xác định (ví dụ: e. 4 phần tử), v.v.

Rõ ràng, thuật toán được trình bày xây dựng cây của giải, tức là xác định phần tử nhỏ nhất của mảng với chính xác $n - 1$ phép so sánh. Nay giờ, thay phần tử nhỏ nhất (nhà vô địch) bằng $-\infty$ trong trang tính tương ứng của nó và tính toán lại nội dung của các đỉnh trên đường lên đỉnh, chúng ta nhận được phần tử lớn nhất tiếp theo ở trên cùng của cây. Vì cây có chiều cao là $\lceil \log_2 n \rceil$, nên quy trình được mô tả để xác định "về nhì", tức là phần tử lớn thứ hai của mảng, yêu cầu $\lceil \log_2 n \rceil - 1$ so sánh thay vì $n - 2$, như Do đó, quá trình sắp xếp mảng hoàn chỉnh yêu cầu không quá $n - 1 + (n - 1)(\lceil \log_2 n \rceil - 1)$ phép so sánh.

Mặc dù thuật toán được trình bày đại diện cho một sự cải tiến thực sự nghiêm trọng liên quan đến thuật toán lựa chọn trực tiếp, nhưng vẫn còn nhiều điều mong muốn. Trước hết, một phương pháp hiệu quả nên được xác định để giảm thiểu số lượng trao đổi. Nhớ lại rằng trao đổi là một hoạt động chậm hơn nhiều so với so sánh và không nên đánh giá thấp vấn đề: chính số lượng trao đổi quá nhiều đã ngăn cản quá trình phân loại rung chuyển (xem 3.2.5) đạt được hiệu quả tiệm cận tốt hơn về phân loại bong bóng (xem ??), mặc dù số lượng so sánh được thực hiện giảm đi rất nhiều. Một nguồn bất tiện khác là các giá trị của nó, dẫn đến việc so sánh không cần thiết, cuối cùng lấp đầy toàn bộ cây. Hơn nữa, thuật toán giải đấu loại trừ, mặc dù nó xây dựng một cây cân bằng hoàn hảo (xem 2.4), không tạo ra một cây mà về mặt lý thuyết có thể mong đợi kết quả tối ưu, vì nó cho phép các lá xuất hiện ở mọi cấp độ. Cuối cùng nhưng không kém phần quan trọng, những người tham gia giải đấu nằm trong danh sách, với những ngọn cây so sánh chỉ sao chép chúng. Điều tự nhiên là muốn giảm thiểu dung lượng bộ nhớ cần thiết, trong thuật toán của giải đấu với loại bỏ có bậc là $2n - 1$. Với mục đích này, chúng ta sẽ cần một cơ chế hiệu quả để tuyển tính hóa cây, cho phép chúng ta chỉ giữ lại những chiếc lá mà không làm mất đi các mối quan hệ đã biết giữa chúng. J. Williams đưa ra giải pháp hiệu quả cho tất cả các vấn đề được nêu ra thông qua cấu trúc kim tự tháp

Định nghĩa 3.1. Ta sẽ gọi một hình chóp là một cây nhị phân cân bằng hoàn hảo với chiều cao h , đồng thời thỏa mãn các điều kiện sau:

- 1) tất cả các lá ở mức h và $h - 1$;
- 2) tất cả những người thừa kế của một đỉnh nhỏ hơn nó;
- 3) tất cả những người thừa kế cấp h đều bị lệch cực đại sang trái.

Điều thú vị ở đây là kim tự tháp cho phép hiện thực hóa hiệu quả không chỉ qua gỗ mà còn thông qua mảng n phần tử, vì các kết nối di truyền giữa các phần tử được định nghĩa là một hàm tuyến tính đơn giản của vị trí của chúng. Phát biểu sau đây cung cấp cho chúng ta một cơ chế cụ thể để thực hiện hiệu suất được mô tả:

Mệnh đề 3.1. *Hình chóp tương đương với dãy khóa h_1, h_{l+1}, \dots, h_r ($1 \leq l \leq r \leq n$), cho mà $h_i \geq h_{2i}$ và $h_i \geq h_{2i+1}$, $i = l, l+1, \dots, r/2$.*

Rõ ràng rằng đỉnh h_1 của hình chóp h_1, h_2, \dots, h_n chứa phần tử lớn nhất của nó. (Tại sao?) Trên thực tế, câu lệnh được cố ý đưa ra dưới dạng tổng quát hơn, dưới đây sẽ nói rõ tại sao. Đặc biệt, tại $l = 1$, sự tương đương là hiển nhiên. Chúng ta sẽ chỉ lưu ý rằng h_{2i} và h_{2i+1} lần lượt là người thừa kế bên trái và bên phải của h_i .

Giả sử rằng chúng ta có các thuật toán hiệu quả để xây dựng một kim tự tháp `buildHeap()`, cũng như để khôi phục một kim tự tháp từ phần tử k , `restoreHeap(k)`, sau khi đỉnh của nó đã được thay thế bằng một phần tử tùy ý. Sau đó, sau khi xây dựng một hình chóp có n phần tử, chúng ta có thể hoán đổi đỉnh h_1 (phần tử lớn nhất của nó) với phần tử cuối cùng h_n , với đỉnh cũ lấy vị trí cuối cùng của nó: cuối cùng trong mảng đã sắp xếp. Vì hoạt động này phá hủy điều kiện của Assertion, nên cần phải sàng lọc đỉnh mới xuống kim tự tháp. Kết quả là ta thu được một hình chóp mới h_1, h_2, \dots, h_{n-1} , chứa ít hơn một phần tử. Tiếp theo là một trao đổi mới của đỉnh của hình chóp với phần tử cuối cùng ($n - 1$) của nó, tiếp theo là một sàng lọc mới. Quá trình được mô tả được lặp lại $n - 1$ lần (không cần đến bước thứ n , vì cuối cùng phần tử thứ n đã ở đúng vị trí, tại sao?). Chúng ta nhận được thuật toán sau để sắp xếp kim tự tháp của một mảng:

```
buildHeap();
for (i = n; i >= 2; i--) {
    swap(m+1, m+i);
    restoreHeap(i-1)
}
```

Làm thế nào để xây dựng kim tự tháp? Nay giờ, hãy giả sử rằng n là số chẵn. Sau đó, theo Mệnh đề cho các phần tử $h_n/2 + 1, h_n/2 + 2, \dots, h_n$, không yêu cầu tỷ lệ bậc nhất vì không có hai phần tử i và j nào được thỏa mãn $j = 2i$ hoặc $j = 2i + 1$. Thật vậy, các yếu tố được đề cập (và chỉ chúng!) Là lá của cây (nằm ở tầng cuối cùng hoặc có thể là áp chót) và không bắt buộc phải thỏa mãn bất kỳ tỷ lệ nào. Hãy mở rộng kim tự tháp bên trái một phần tử. Phần tử x mới được thêm vào sẽ ở mức áp chót và sẽ có ít nhất một phần tử kế nhiệm. Điều này đòi hỏi phải kiểm tra để xem liệu các mối quan hệ trong Yêu cầu bồi thường có bị vi phạm hay không, sau đó là trao đổi x với một trong hai người thừa kế của anh ta. Nay giờ các phần tử $h_n/2, h_n/2 + 1, \dots, h_n$ lại tạo thành một hình chóp. Tiếp theo là sự mở rộng mới của kim tự tháp bên trái, và quá trình này tiếp tục cho đến khi tất cả các yếu tố được bao gồm. Đôi với n là lẻ là $h_n/2 + 1, h_n/2 + 2, \dots, h_n$. Trong thực tế, trong việc triển khai C không cần phải phân biệt giữa hai trường hợp, vì phép toán / là một số nguyên.

Chúng ta sẽ lưu ý rằng việc sàng lọc phần tử mới được thêm vào x xuống kim tự tháp có thể diễn ra trong nhiều giai đoạn, tức là sau khi trao đổi x với một trong những phần tử kế nhiệm của nó, nó có thể trở lại rằng x không đáp ứng các điều kiện của Mệnh đề, v.v. Vì vậy, x được sàng xuống kim tự tháp, chìm xuống một bậc ở mỗi bậc cho đến khi vị trí của nó. Một cải tiến rõ ràng có thể được thực hiện ở đây, đẩy nhanh quá trình, cụ thể là trao đổi x với nhỏ hơn của hai người kế nhiệm, mặc dù điều này không phải lúc nào cũng có lợi. Ngoài ra, việc trao đổi x với những người kế nhiệm của nó có thể bị hoãn lại cho đến khi tìm được vị trí chính xác của nó và thay vào đó, các bài toán một chiều được thực hiện trong quá trình tìm kiếm vị trí cuối cùng của nó.

Lưu ý rằng cả việc xây dựng và trùng tu đều yêu cầu sàng lọc phần tử ở đỉnh kim tự tháp xuống kim tự tháp. Điều này dẫn chúng ta đến ý tưởng triển khai một hàm sàng lọc sift () chung:

Sàng lọc phần tử (trong 313heapsort.c)

```

/* Sàng lọc phần tử từ trên xuống của kim tự tháp */
void sift (struct CElem m[], unsigned l, unsigned r)
{ unsigned i = l, j = i + i;

```

```

struct CElem x = m[i];
while (j <= r) {
    if (j < r && m[j].key < m[j+1].key)
        j++;
    if (x.key >= m[j].key)
        break;
    m[i] = m[j];
    i = j;
    j <= 1; /*tương đương với j*=2; */
}
m[i] = x;
}

```

Bây giờ việc xây dựng kim tự tháp (hàm buildHeap()) có thể được thực hiện bằng cách sử dụng đoạn chương trình sau:

```

for (k = n/2 + 1; k > 1; k--) {
    sift (m,k-1,n);
}

```

Để sắp xếp nó, liên tục hoán đổi đỉnh của kim tự tháp với phần tử cuối cùng của nó, tiếp theo là sàng lọc (chức năng restoreHeap()), chúng ta nhận được như sau:

```

for (k = n; k > 1; k--) {
    swap(m+1,m+k);
    sift (m,1,k-1);
}

```

Cuối cùng, để sắp xếp hình chóp, chúng ta nhận được:

Chương trình 3.13. Sắp xếp kiểu kim tự tháp (313heapsort.c)

```

void heapSort(struct CElem m[], unsigned n) /*Sắp xếp kim tự tháp
*/
{
    unsigned k;
    /* 1. Xây dựng kim tự tháp*/
    for (k = n/2 + 1; k > 1; k--) {
        sift (m,k-1,n);
    }
    /* 2. Xây dựng một chuỗi được sắp xếp*/
    for (k = n; k > 1; k--) {
        swap(m+1,m+k);
    }
}

```

```

    sift (m,1,k-1);
}
}

```

Bài tập

- ▷ 3.25. Chứng minh rằng trong một giải đấu quần vợt loại trực tiếp, nếu số người tham gia không phải là cấp 2 thì ít nhất trong một hiệp sẽ phải có một đấu thủ nghỉ.
- ▷ 3.26. Chứng minh rằng đỉnh của hình chóp chứa phần tử lớn nhất của nó.
- ▷ 3.27. Phát triển một phương pháp sắp xếp bằng cách sử dụng một kim tự tháp bậc ba. Kim tự tháp bậc ba là một bản tóm tắt của kim tự tháp cổ điển và dựa trên các cây bậc ba (ba ngôi) hoàn chỉnh. Chúng ta có nên mong đợi gia tốc so với kim tự tháp cổ điển?
- ▷ 3.28. Để phát triển một thuật toán để nhanh chóng hợp nhất hai kim tự tháp.

3.2.10. Độ phức tạp về thời gian tối thiểu của việc sắp xếp theo cách so sánh

Tất cả các thuật toán được xem xét cho đến nay đều thuộc loại thuật toán sắp xếp so sánh, tức là những thuật toán trong đó phép toán duy nhất được phép là so sánh giữa các cặp phần tử sử dụng các phép toán $>$ (\geq), $<$ (\leq) và $=$ (\neq). Đơn giản nhất trong số chúng (ví dụ: chèn, bong bóng hoặc lựa chọn trực tiếp, xem 3.2.2, 3.2.4 và 3.2.8) thực hiện $\Theta(n^2)$ số phép so sánh, trong khi phép so sánh tốt nhất có độ phức tạp $\Theta(n \log_2 n)$ trong ở giữa (phân loại nhanh, xem 3.2.6) hoặc thậm chí trong trường hợp xấu nhất (phân loại theo hình chóp: xem 3.2.9, phương pháp thỏ và rùa: xem 3.2.7 và phân loại theo hợp nhất, sẽ được xem xét trong 7.3.). Đối với mỗi thuật toán như vậy, một trình tự đầu vào có thể được tìm thấy để đạt được các ước tính này, tức là các thuật toán sắp xếp tốt nhất mà chúng ta biết cho đến nay có độ phức tạp thuật toán $\Theta(n \log_2 n)$. Có thể không cải thiện đánh giá này? câu trả lời cho câu hỏi này như sau:

Mệnh đề 3.2. Mọi thuật toán sắp xếp so sánh thực hiện một số phép so sánh bậc của $\Theta(n \log_2 n)$.

Bằng chứng. Đối với mỗi thuật toán sắp xếp so sánh, một cây so sánh nhị phân tương ứng có thể được so sánh duy nhất, tương tự như trong Hình 3.1. Những chiếc lá là $n!$ theo số lượng và mỗi trong số chúng chứa chính xác một trong các hoán vị của chuỗi đầu vào. Các tờ rơi, đến lượt nó, chứa các so sánh giữa một cặp yếu tố. Không giới hạn cộng đồng, chúng ta có thể giả định rằng tất cả các so sánh đều thuộc loại $<$. Thật vậy, với mỗi quan hệ $=, \neq, \leq, <, >$ và \geq chúng ta có rằng nó hợp lệ hoặc không, nghĩa là mỗi phép so sánh như vậy có hai đầu ra, tương ứng với hai người thừa kế trong cây. Việc thực thi thuật toán sắp xếp phụ thuộc vào việc tìm đường dẫn trong cây nhị phân. Việc tìm kiếm bắt đầu từ gốc và kết thúc khi đạt đến lá. Hoán vị có trong trang tính xác định thứ tự cần thiết của chuỗi đầu vào.

Như đã đề cập ở trên, độ phức tạp của thuật toán sắp xếp ở mức tối tệ nhất được xác định bởi số lượng phép so sánh tối đa mà nó thực hiện, tức là độ dài của đường dẫn tối đa từ gốc đến một trong các lá. Để sắp xếp đúng mỗi dây đầu vào của n phần tử, thuật toán phải có kết quả là mỗi trong số n có thể! hoán vị, tức là mỗi hoán vị phải xuất hiện dưới dạng một lá trong cây so sánh của nó ít nhất một lần.

Gọi h là chiều cao của cây (tức là chiều dài của con đường tối đa từ gốc đến một trong các lá của nó). Bất đẳng thức là hợp lệ:

$$n! \leq 2^h.$$

Từ đây sau khi tính logarit ta được: $h \geq \log_2(n!)$

Để logarit $n!$, chúng ta sẽ sử dụng xấp xỉ Stirling $n! > (n/e)^n$, nơi chúng ta nhận được:

$$h \geq \log_2(n!) \geq \log_2(n/e)^n = n \cdot \log_2 n - n \cdot \log_2 e,$$

khi nó trực tiếp theo sau $h \in \Omega(n \log_2 n)$. Nghĩa là, chiều cao của cây, và do đó số phép so sánh bắt buộc tối thiểu, là $\Omega(n \log_2 n)$.

Ngay sau phát biểu trên rằng phân loại theo hình chóp, phương pháp của thỏ và rùa và sắp xếp theo tiệm cận là tối ưu về mặt tiệm

cận. Ở đây, chúng ta sẽ bảo lưu rằng trong một số điều kiện hạn chế mạnh hơn (chủ yếu liên quan đến việc phân phối các giá trị khóa và / hoặc đại diện bên trong của chúng) thì sẽ có các thuật toán hiệu quả hơn, một số thuật toán trong số đó chúng ta sẽ xem xét trong đoạn 3.2 tiếp theo.

3.2.11. Bài tập

- ▷ 3.29. Chứng minh rằng mọi tập hợp con của một tập hợp có phép lệnh tuyến tính cũng được sắp xếp theo thứ tự tuyến tính.

3.3. Sắp xếp theo sự biến đổi

Có một loại thuật toán rộng khác không dựa trên so sánh mà dựa trên việc thực hiện các phép toán (chủ yếu là số học) trên các phần tử của tập hợp - *sắp xếp theo phép biến đổi*. Sự khác biệt giữa hai cách tiếp cận có thể so sánh với sự khác biệt giữa tìm kiếm nhị phân (xem 2.3) và băm (xem 2.5). Thông thường, các thuật toán sắp xếp chuyển đổi chỉ có thể áp dụng trong một số trường hợp cụ thể dưới các điều kiện hạn chế bổ sung liên quan đến loại và / hoặc tập hợp các giá trị cho phép và / hoặc số lần xuất hiện chính của các phần tử được sắp xếp.

3.3.1. Sắp xếp theo tập hợp

Đại diện đầu tiên của loại thuật toán sắp xếp theo phép biến đổi, mà chúng ta sẽ xem xét, là thuật toán sắp xếp theo tập hợp. Nó có thể áp dụng cho một tập hợp có hàm thứ tự f thỏa mãn đồng thời các điều kiện sau (Ta ký hiệu tập hợp là M và số phần tử của nó là n):

1) các giá trị của f là các số tự nhiên trong khoảng $[a, b]$ đã cho, chứa $m = b - a + 1$ số nguyên. Không phải mọi số nguyên từ $[a, b]$ đều phải là giá trị của f , nhưng mọi giá trị của f phải từ $[a, b]$.

2) f là đơn ánh, nghĩa là với $x_1, x_2 \in M$ và $x_1 \neq x_2$, chúng ta có $f(x_1) \neq f(x_2)$ (tức là không lặp lại).

Để đơn giản hóa thuật toán, chúng ta sẽ không làm việc với các phần tử có kiểu **struct** CElem, nhưng chúng ta sẽ sử dụng kiểu **unsigned**, giả sử rằng giá trị của phần tử lớn nhất không vượt quá

một hằng số hợp lý MAX_VALUE. (Trên thực tế, MAX_VALUE là ký hiệu chương trình của m) Đây là cách chúng ta sẽ làm việc với khoảng $[0, MAX_VALUE]$. Việc sắp xếp được thực hiện một cách tuyến tính bằng cách chuyển một qua các phần tử của mảng, sau đó chuyển qua các số trong khoảng. Lúc đầu, chúng ta khởi tạo với 0 các phần tử của set[] (kiểu char, chúng ta sẽ coi là Boolean), sau đó chúng ta đi qua mảng và với mỗi phần tử của nó, chúng ta đặt 1 phần tử tương ứng của tập . Trong bước thứ hai, chúng ta kiểm tra tuần tự tất cả các số trong khoảng thời gian xem chúng có thuộc set[] hay không. Ở đầu lần vượt qua thứ hai, chúng ta coi rằng $m[]$ không chứa bất kỳ phần tử nào và chúng ta xây dựng lại nó, lần này là một dãy đã được sắp xếp. Các chi tiết có thể được nhìn thấy từ đoạn chương trình đính kèm.

Chương trình 3.14. Sắp xếp theo tập hợp (314setsort.c)

```

void setSort(unsigned m[], unsigned n)
{
    char set[MAX_VALUE];
    unsigned i,j;

    /* 0.Khởi tạo tập hợp*/
    for (i = 0; i < MAX_VALUE; i++)
        set[i] = 0;

    /* 1. Sự hình thành của tập hợp*/
    for (j = 0; j < n; j++) {
        assert(m[j] >= 0 && m[j] < MAX_VALUE);
        assert(0 == set[m[j]]));
        set[m[j]] = 1;
    }

    /* 2.Tạo trình tự đã sắp xếp */
    for (i = j = 0; i < MAX_VALUE; i++)
        if (set[i])
            m[j++] = i;
        assert(j == n);
}

```

Độ phức tạp của thuật toán trên là $\Theta(m + n)$ và đối với các giá trị của n gần với m , nó có thể được coi là tuyến tính đối với n . Cần lưu

ý rằng điều này không phải là "miễn phí": Một mặt, chúng ta có một giả định bổ sung nặng nề về các giá trị và loại khóa của các phần tử được sắp xếp (ví dụ: chúng ta không thể sắp xếp các số thực), và mặt khác - chúng ta sử dụng bộ nhớ bổ sung của $\Theta(m)$. Chúng ta sẽ lưu ý rằng hầu hết các thuật toán được thảo luận ở trên đều sử dụng bộ nhớ bổ sung không đổi. Một số thuật toán đệ quy, chẳng hạn như sắp xếp nhanh (xem 3.2.6), yêu cầu bộ nhớ logarit và trong trường hợp xấu nhất là thậm chí bộ nhớ tuyến tính (điều này bị ẩn đằng sau cơ chế đệ quy). Tuy nhiên, có một yêu cầu lớn hơn ở đây: không phải $\Theta(n)$, còn $\Theta(m)$, vì $m > n$, và $m \gg n$ là có thể (tức là m có thể lớn hơn n nhiều).

Chúng ta có thể tóm tắt thuật toán, mở rộng phạm vi của nó. Chúng ta sẽ "thoát khỏi" sự hạn chế khó chịu của việc chỉ làm việc với các số nguyên. Lần này chúng ta sẽ sử dụng các phần tử kiểu **struct** CElem mà khóa thỏa mãn hai điều kiện trên. Độ phức tạp của thuật toán này một lần nữa sẽ là $\Theta(m + n)$, trong đó m là số nguyên trong khoảng $[a, b]$.

Lần này khi tổ chức tập hợp, chúng ta sẽ phải giữ thông tin không chỉ về việc một số từ $[a, b]$ có xảy ra như một khóa của một phần tử của m[] hay không, mà còn cả giá trị của phần tử này. Nhìn chung, có hai cách tiếp cận về vấn đề này. Trong cách tiếp cận đầu tiên, chỉ số của phần tử tương ứng trong m[] được giữ trong tập hợp:

```
struct CSetEl {
    char found;
    unsigned index;
} m[MAX];
```

Trường tìm được có thể được lưu bằng cách giả sử rằng phần tử rỗng của m[] không được sử dụng. Số không có nghĩa là không có phần tử có khóa như vậy và bất kỳ giá trị nào khác không sẽ là chỉ số trong m[] của phần tử có khóa tương ứng:

unsigned m[MAX];

Nhân tiện, từ quan điểm của ngôn ngữ của chúng ta, tốt hơn là sử dụng cho giá trị chính thức không phải là 0, mà ngược lại - giá trị lớn nhất của kiểu không dấu. Đây là một giá trị khác nhau cho các nền tảng khác nhau (ví dụ: trong DOS là 65,535 và trong Windows

là 4,294,967,295). Chúng ta sẽ sử dụng cách tiếp cận được đề cập trong Chương 1 để lấy nó - dựa trên sự biểu diễn bên trong của các số trong mã bổ sung: (`unsigned`)(-1).

Ưu điểm của hai khai báo trên là tính kinh tế, trong điều kiện bộ nhớ được sử dụng, không phải các phần tử của `m[]` được bảo toàn, có thể là các bản ghi kích thước lớn, nhưng chỉ các chỉ số của chúng. Lưu ý hai (các) xác nhận bảo vệ chúng ta khỏi các giá trị khóa trùng lặp và không hợp lệ.

Chương trình 3.15. Sắp xếp theo tập hợp (315setsort2.c)

```
#define NO_INDEX (unsigned)(-1)
/*Sắp xếp một mảng bằng cách sử dụng tập hợp*/
void setSort(struct CElem m[], unsigned n)
{
    unsigned indSet[MAX_VALUE]; /*Bộ chỉ mục*/
    unsigned i,j;

    /* 0. Khởi tạo tập hợp*/
    for (i = 0; i < MAX_VALUE; i++)
        indSet[i] = NO_INDEX;

    /* 1.Sự hình thành của bộ*/
    for (j = 0; j < n; j++) {
        assert(m[j].key >= 0 && m[j].key < MAX_VALUE);
        assert(NO_INDEX == indSet[m[j].key]);
        indSet[m[j].key] = j;
    }

    /* 2.Tạo trình tự đã sắp xếp*/
    for (i = j = 0; i < MAX_VALUE; i++)
        if (NO_INDEX != indSet[i])
            do4Elem(m[indSet[i]]);
}
```

Chương trình đính kèm giải quyết một phần vấn đề. Nó nhận được chuỗi các phần tử đã được sắp xếp, nhưng không có trong mảng `m[]`. Thay vào đó, nó gọi hàm `do4Elem()` cho mỗi mục tiếp theo trong chuỗi đã sắp xếp. Điều này có thể chấp nhận được nếu chúng ta muốn thực hiện một thao tác trên các phần tử một lần. Ví dụ: để hiển thị chúng ở dạng đã sắp xếp.

Việc duy trì trình tự đã sắp xếp sẽ hủy bỏ $m[]$, và do đó các chỉ số tương ứng. Một cách khả thi là lưu trước một bản sao của $m[]$. Chúng ta có thể tự giải quyết rắc rối khi duy trì một bản sao bằng cách giữ các phần tử tương ứng của $m[]$ trong tập hợp. Tất nhiên, phần sau có liên quan đến sự lãng phí bộ nhớ bổ sung - một mục nhập kiểu **struct** CElem cho mỗi số trong khoảng $[a, b]$. Có tính đến kích thước của các bản ghi đã sắp xếp là cố định, chúng ta nhận được rằng kích thước của bộ nhớ bổ sung cần thiết một lần nữa là $\Theta(m)$, lần này đằng sau ký hiệu $\Theta(\dots)$ có một hằng số lớn hơn.

Bài tập

- 3.30. Sửa đổi việc triển khai nhiều sắp xếp được đề xuất để thu được trình tự đã sắp xếp trong mảng đầu ra.

3.3.2. Sắp xếp theo số đếm

Chúng ta hãy cố gắng phát triển hơn nữa ý tưởng về đa phân loại. Nhược điểm chính của phương pháp này là yêu cầu về tính duy nhất của khóa, tức là không được phép lặp lại. Giả sử chúng ta muốn sắp xếp các phần tử của một tập hợp các số tự nhiên từ một khoảng cho trước, mỗi phần tử có thể xảy ra không quá k lần. Ở đây các phần tử của mảng không thuộc kiểu bản ghi và vai trò của khóa được đảm nhận bởi danh tính, tương tự như trường hợp sắp xếp các số nguyên theo một tập hợp. Chúng ta định nghĩa một mảng $cnt[]$, trong đó $cnt[i]$ chứa số lần xuất hiện của số i . Chúng ta bắt đầu bằng cách đặt lại các phần tử của $cnt[]$, sau đó đi qua các phần tử của $m[]$ và tìm số lần xuất hiện cho mỗi số nguyên của $[a, b]$. Trong bước thứ hai, chúng ta đi qua các phần tử của $cnt[]$, bao gồm trong dãy đã sắp xếp $cnt[x]$ số x , với mỗi số nguyên $x \in [a, b]$.

Chương trình 3.16. Sắp xếp theo số đếm (316counts.c)

```
void countSort(unsigned m[], unsigned n) /* Sắp xếp theo số đếm */
{
    unsigned char cnt[MAX_VALUE];
    unsigned i,j;
    /* 0.Khởi tạo tập hợp*/
    for (i = 0; i < MAX_VALUE; i++)
        cnt[i] = 0;
    /* 1.Sự hình thành của bộ*/
}
```

```

for (j = 0; j < n; j++) {
    assert(m[j] >= 0 && m[j] < MAX_VALUE);
    cnt[m[j]]++;
}
/* 2. Tạo trình tự đã sắp xếp */
for (i = j = 0; i < MAX_VALUE; i++)
while (cnt[i]--)
    m[j++] = i;
    assert(j == n);
}

```

Thuật toán trên hoạt động tốt khi giả sử rằng mỗi số xảy ra không quá k lần, đối với một hằng số k xác định trước hợp lý. Về mặt lý thuyết, lĩnh vực ứng dụng của thuật toán trên vẫn bị giới hạn bởi các yêu cầu sau:

- các phần tử phải nằm trong khoảng $[a, b]$, và không phải mọi số từ $[a, b]$ đều phải là giá trị của f , mà tất cả các giá trị của f phải từ $[a, b]$.
- Mỗi số $[a, b]$ xảy ra không quá k lần;
- Chỉ sắp xếp các số, không phải các mục nhập ngẫu nhiên.

Hai hạn chế đầu tiên của thuật toán trên là tự nhiên. Nhân tiện, những hạn chế như vậy là đặc trưng của hầu hết các thuật toán sắp xếp theo phép biến đổi. Tuy nhiên, yêu cầu thứ hai không quá cơ bản và chúng ta có thể dễ dàng loại bỏ nó. Vấn đề chính là cần phải giữ thông tin không chỉ về việc một số nguyên x của $[a, b]$ có xảy ra như một khóa của một phần tử của M hay không, mà còn là giá trị của phần tử này. Chúng ta giải quyết vấn đề tương đối dễ dàng: chúng ta sẽ giữ thông tin về việc khóa x có xảy ra hay không và ngoài ra - phần tử tương ứng với khóa x .

Khi sắp xếp bằng cách đếm cho mỗi giá trị cho phép của khóa, có thể có một số yêu tố liên quan, điều này làm phức tạp vấn đề: cần phải tổ chức một danh sách. Vì số lần xuất hiện của một khóa không vượt quá k nên mỗi danh sách như vậy có thể chứa nhiều nhất k phần tử. Tuy nhiên, việc cấp phát bộ nhớ tĩnh có kích thước k cho mỗi khóa p dẫn đến sự lãng phí không cần thiết.

Trong chương trình dưới đây, mỗi khóa được liên kết với một danh sách động chứa các phần tử tương ứng của M . Việc duy trì

một con trỏ duy nhất ở đầu danh sách có thể là một nguồn tiềm ẩn của các vấn đề, vì nó làm cho việc sắp xếp không bền vững (danh sách không được liên kết). Do đó, các phần tử có cùng khóa rơi vào trình tự ban đầu theo thứ tự ngược lại với thứ tự mà chúng có trong M. Phần tử sau có thể dễ dàng được sửa chữa, ví dụ bằng cách thêm một con trỏ bổ sung vào cuối danh sách (hoặc để báo cáo sau), khi tập hợp dãy đã sắp xếp: lật tất cả các danh sách). Độ phức tạp của thuật toán lại là $\Theta(m + n)$.

Chương trình 3.17. Sắp xếp theo số đếm (317counts2.c)

```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define MAX 100
struct CElem {
    int key;
    /* .....Một số dữ liệu ..... */
};
struct CList {
    struct CElem data;
    struct CList *next;
};

struct CList* init (unsigned n) /* Điền vào mảng với các số ngẫu nh
    ên*/
{
    struct CList *head, *p;
    unsigned i;
    srand(time(NULL));
    for (head = NULL, i = 0; i < n; i++) {
        p = (struct CList *) malloc(sizeof(struct CList));
        p->data.key = rand();
        assert(p->data.key);
        p->next = head;
        head = p;
    }
    return head;
}

struct CList *bitSort(struct CList *head)
```

```
{ struct CList *zeroEnd, *oneEnd, *zero, *one;
  unsigned maxBit, bitPow2;
  /* 0. Xác định mặt nạ bit tối đa */
  maxBit = 1 << (8*sizeof(head->data.key)-1);

  /* 1. phần tử giả ở đầu danh sách */
  zero = (struct CList *) malloc(sizeof(struct CList));
  one = (struct CList *) malloc(sizeof(struct CList));

  /* 2. Sắp xếp */
  for (bitPow2 = 1; bitPow2 < maxBit; bitPow2 <<= 1) {

    /* 2.1. Phân phối theo danh sách*/
    for (zeroEnd = zero, oneEnd = one; NULL != head; head = head->
next)
      if (!(head->data.key & bitPow2)) {
        zeroEnd->next = head;
        zeroEnd = zeroEnd->next;
      }
      else {
        oneEnd->next = head;
        oneEnd = oneEnd->next;
      }
    /* 2.2. Hợp nhất danh sách */
    oneEnd->next = NULL;
    zeroEnd->next = one->next;
    head = zero->next;
  }

  /* 3. Giải phóng bộ nhớ */
  free(zero);
  free(one);
  return head;
}

void print(struct CList *head)
{ for (; NULL != head; head = head->next)
  printf("%8d", head->data.key);
  printf("\n");
}
```

```

void check(struct CList *head)
{ if (NULL == head) { return; }
  for (; NULL != head->next; head = head->next)
    assert(head->data.key <= head->next->data.key);
}
void clear(struct CList *head)
{ struct CList *p = head;
  while (NULL != head) {
    head = (p = head)->next;
    free(p);
  }
}

int main()
{
  struct CList *head;
  head = init(MAX);
  printf("Mảng trước khi sắp xếp:\n"); print(head);
  head = bitSort(head);
  printf("Mảng sau khi sắp xếp :\n"); print(head);
  check(head);
  clear(head);
  return 0;
}

```

Bài tập

- ▷ 3.31. Triển khai một phương án bền vững của sắp xếp bằng cách đếm.

3.3.3. Sắp xếp theo bit

Bất chấp những cải tiến đã được thực hiện, sắp xếp theo số đếm (xem) kẽ thừa hầu hết những thiếu sót của sắp xếp theo nhiều (xem) và khó có thể khẳng định là ít nhất một phần phổ quát. Vấn đề chính của thuật toán là cần phải duy trì một mảng bổ sung với kích thước tỷ lệ với số giá trị có thể có của các khóa m. Rõ ràng rằng m có thể là một số đủ lớn và không cho phép cấp phát lượng bộ nhớ cần thiết. Trong thực tế, hầu hết các khóa có thể có có thể không được tìm thấy, trong khi đối với các khóa khác, có thể có sự tích tụ đáng kể của các phần tử, dẫn đến việc sử dụng bộ nhớ không hiệu quả.

Sắp xếp bằng cách đếm chuyển một lần qua các phần tử của tập hợp, yêu cầu thời gian $\Theta(n)$ và một lần - qua tập các giá trị cho phép đổi với các khóa, trong thời gian $\Theta(m)$. Tổng độ phức tạp của thuật toán trở thành $\Theta(m + n)$. Kết quả thu được cho thấy thuật toán là tuyến tính đối với m và n . Sau đó nó là tuyến tính như thế nào? Rõ ràng rằng đối với $m > n$, thuật toán sắp xếp theo phép đếm là tuyến tính với số giá trị có thể có của các khóa m . Tuy nhiên, đối với chúng ta, điều quan trọng hơn là nó phức tạp như thế nào về số phần tử n . Không khó để nhận thấy rằng đối với $m = n^3$, độ phức tạp của thuật toán trở thành $\Theta(n^3)$, tức là nó thậm chí còn tệ hơn việc sắp xếp theo phương pháp bong bóng! Và nếu $m = n^7$? Chúng ta nhận được sự thiếu hiệu quả tuyệt vọng ...

Mặc dù thoạt nhìn điều này có vẻ không ổn, nhưng ý tưởng sắp xếp bằng cách đếm rất hay và có thể đưa chúng ta đến một thuật toán sắp xếp tuyến tính thực sự. (Tất nhiên, một lần nữa với một số hạn chế ...) Đầu tiên chúng ta sẽ tập trung vào thuật toán sắp xếp bit, sau đó chúng ta sẽ xem xét tóm tắt tự nhiên của nó - phương pháp hệ thống số (xem 3.3.4).

Ý tưởng về sắp xếp theo bit chủ yếu dựa trên biểu diễn bên trong nhị phân của các số trong máy tính. Hãy đặt một tập hợp các số nguyên không dấu mà chúng ta sẽ sắp xếp các phần tử. Chúng ta chia các số thành hai danh sách tùy thuộc vào giá trị của chữ số nhị phân nhỏ nhất của chúng. Các số chẵn nằm trong danh sách đầu tiên, và các số lẻ - trong danh sách thứ hai. Sau đó, chúng ta thêm danh sách thứ hai vào cuối danh sách đầu tiên, nơi chúng ta nhận được một danh sách chung. Không giống như đếm, mục mới phải ở cuối danh sách. Ý nghĩa của việc làm sáng tỏ này sẽ trở nên rõ ràng ở phần sau. Lặp lại thao tác với bit áp chót, sau đó với bit áp chót, v.v. Quá trình kết thúc sau khi thực hiện thao tác với bit cao nhất. Không khó để thấy rằng quá trình được mô tả theo cách này thực sự dẫn đến một trình tự được sắp xếp (Tại sao?). Yêu cầu trên bây giờ đã rõ ràng. Nó đảm bảo với chúng ta rằng mỗi bước tiếp theo sẽ được hưởng lợi từ những bước trước đó và trong trường hợp các giá trị bằng nhau của bit tương ứng, nó sẽ giữ nguyên lệnh thu được trong các bước trước đó. Vì vậy, mỗi bước trở nên bền vững, và do đó phương pháp là bền vững.

Dựa trên biểu diễn bên trong của dữ liệu trong máy tính, không khó để giả định rằng phương pháp được mô tả có thể áp dụng, với những sửa đổi tối thiểu, để sắp xếp bất kỳ kiểu dữ liệu nào - sắp xếp các cấu trúc phức tạp hơn như mảng, bản ghi và chuỗi thực tế được giảm thành sắp xếp trong số các con số. Tuy nhiên, cần lưu ý rằng điều này không phải lúc nào cũng đơn giản như vậy. Ví dụ: có một số vấn đề với việc sắp xếp các số âm (được trình bày nội bộ trong mã bổ sung) cũng như các số dấu phẩy động.

Chương trình 3.18. Sắp xếp theo bit (318bitsort.c)

```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define MAX 100
struct CElem {
    int key;
    /* ..... Dữ liệu khác ..... */
};

struct CList {
    struct CElem data;
    struct CList *next;
};

struct CList* init (unsigned n) /*Điền vào mảng với các số ngẫu nhiê
n*/
{
    struct CList *head, *p;
    unsigned i;
    srand(time(NULL));
    for (head = NULL, i = 0; i < n; i++) {
        p = (struct CList *) malloc(sizeof(struct CList));
        p->data.key = rand();
        assert(p->data.key);
        p->next = head;
        head = p;
    }
    return head;
}

struct CList *bitSort(struct CList *head)
```

```

{ struct CList *zeroEnd, *oneEnd, *zero, *one;
  unsigned maxBit, bitPow2;
  /* 0.Xác định mặt nạ bit tối đa */
  maxBit = 1 << (8*sizeof(head->data.key)-1);
  /* 1. Phần tử giả ở đầu danh sách*/
  zero = (struct CList *) malloc(sizeof(struct CList));
  one = (struct CList *) malloc(sizeof(struct CList));
  /* 2. Sắp xếp*/
  for (bitPow2 = 1; bitPow2 < maxBit; bitPow2 <= 1) {
    /* 2.1. Phân phối theo danh sách*/
    for (zeroEnd = zero, oneEnd = one; NULL != head; head = head->
        next)
      if (!(head->data.key & bitPow2)) {
        zeroEnd->next = head;
        zeroEnd = zeroEnd->next;
      }
    else {
      oneEnd->next = head;
      oneEnd = oneEnd->next;
    }

    /* 2.2. Hợp nhất danh sách*/
    oneEnd->next = NULL;
    zeroEnd->next = one->next;
    head = zero->next;
  }
  /* 3. Giải phóng bộ nhớ*/
  free(zero);
  free(one);
  return head;
}

void print(struct CList *head)
{ for (; NULL != head; head = head->next)
  printf("%8d", head->data.key);
  printf("\n");
}

void check(struct CList *head)
{ if (NULL == head) { return; }

```

```

for (; NULL != head->next; head = head->next)
    assert(head->data.key <= head->next->data.key);
}

void clear(struct CList *head)
{
    struct CList *p = head;
    while (NULL != head) {
        head = (p = head)->next;
        free(p);
    }
}

int main()
{
    struct CList *head;
    head = init(MAX);
    printf("Mảng trước khi sắp xếp:\n"); print(head);
    head = bitSort(head);
    printf("Mảng sau khi sắp xếp:\n"); print(head);
    check(head);
    clear(head);
    return 0;
}

```

Trong cách triển khai trên, dữ liệu được tổ chức thành các danh sách động. Hiệu quả tốt hơn có thể đạt được khi triển khai tĩnh. Với mục đích này, một mảng n phần tử bổ sung duy nhất là đủ, trong đó hai danh sách phát triển so với nhau. Điều này yêu cầu sửa đổi quá trình duyệt qua các phần tử của mảng, bởi vì, mặc dù chúng hợp nhất, hai danh sách vẫn tách biệt. Trong bước tiếp theo, danh sách sẽ phát triển từ cả hai đầu của mảng ban đầu, v.v. Cuối cùng, một bước bổ sung sẽ được yêu cầu để đảo ngược danh sách thứ hai, sẽ yêu cầu không quá $n/2$ lần trao đổi (trong trường hợp xấu nhất, danh sách thứ hai chứa tất cả các phần tử). [Shishkov-1995] [ComputerNews-1994a] [TopTeam-1997]

Không khó để nhận thấy rằng sắp xếp theo bit là tuyến tính và có độ phức tạp của bậc $C.n$, trong đó C là số chữ số của các số được sắp xếp. Trong trường hợp chung, trong triển khai động, bộ nhớ được yêu cầu bổ sung là không đổi.

Thuật toán có thể được đảo ngược và buộc phải xem các bit từ

già nhất đến trẻ nhất. Ý tưởng là, giống như cách sắp xếp nhanh của Hoor, chia mảng thành hai phân vùng: bên trái - với bit 0 cao nhất và bên phải - với bit 1 cao nhất, rồi áp dụng thao tác tương tự cho từng phân vùng, như lần này sự phân chia diễn ra theo chút thâm niên tiếp theo. Phép chia đệ quy kết thúc sau khi xem xét bit nhỏ nhất. Không giống như cách sắp xếp nhanh Hoor cổ điển, số 2^b , $b = 0, 1, 2, \dots$ ở đây được sử dụng như một dấu phân cách. Vì 2^b không nhất thiết phải chứa trong tập nguồn, nên không có gì đảm bảo rằng trong lần lặp hiện tại, một phần tử sẽ đi đến vị trí cuối cùng của nó. Do đó, đối với $i = j$, có thể xảy ra trao đổi dư thừa các phần tử. (Tại sao?) Sau đây là một ví dụ thực hiện.

Chương trình 3.19. Sắp xếp theo bit (319bitsort2.c)

```

struct CElem m[MAX];
.....
void bitSort2(int l, int r, unsigned bitMask)
{
    int i, j;
    if (r > l && bitMask > 0) {
        i = l; j = r;
        while (j != i) {
            while (!(m[i].key & bitMask) && i < j) i++;
            while ((m[j].key & bitMask) && j > i) j--;
            swap(&m[i], &m[j]);
        }
        if (!(m[r].key & bitMask)) j++;
        bitSort2(l, j-1, bitMask >> 1);
        bitSort2(j, r, bitMask >> 1);
    }
}
.....
int main() {
    .....
    bitSort2(0, MAX-1, 1 << 8 * sizeof(m[0].key) - 1);
    .....
    return 0;
}

```

Phiên bản đệ quy của sắp xếp bit có một hành vi thú vị: mảng càng xáo trộn, nó càng hoạt động tốt hơn và với một mảng được xáo trộn nhiều, nó thậm chí còn vượt trội theo tốc độ phân loại nhanh

của Hoor.

Các triển khai trên là ví dụ và giả định rằng các giá trị chính được phân phối tương đối đồng đều trên phạm vi. Trong thực tế, điều này không phải luôn luôn như vậy. Giả sử chúng ta giả định các giá trị chính trong phạm vi 0-65535, trong khi thực tế chúng nằm trong 0-1000. Do đó, khi áp dụng sắp xếp bit, 6 bước cuối cùng của thuật toán trở nên thừa, vì chúng không thay đổi thứ tự của các phần tử theo bất kỳ cách nào: tất cả các số đều có 0 trong bit tương ứng.

bài tập

- ▷ 3.32. Chứng minh rằng việc sắp xếp bit sẽ sắp xếp từng chuỗi đầu vào một cách chính xác.
- ▷ 3.33. Để triển khai một biến thể tĩnh của sắp xếp bit, trong đó một mảng bổ sung duy nhất được sử dụng, trong đó hai danh sách, được xây dựng trong quá trình triển khai lặp lại, tăng lên so với nhau.
- ▷ 3.34. Để so sánh biến thể lặp lại và đệ quy của sắp xếp bit.

3.2.4.

3.3.4. Phương pháp hệ đếm số

Thuật toán trên cho kết quả xuất sắc. Lần đầu tiên, chúng ta triển khai một thuật toán sắp xếp tuyển tính thực sự, giảm các yêu cầu bổ sung nhiều nhất có thể. Chưa hết . . . Điều gì sẽ xảy ra nếu chúng ta sử dụng không phải hai mà là ba danh sách? Và tại sao không phải là 10? Hay 16? Hoặc thậm chí 256? Rõ ràng, khi số lượng danh sách tăng lên, hằng số C sẽ giảm, tức là số lần đi qua các phần tử của tập hợp.

Ý tưởng của phương pháp hệ đếm số tương tự như phương pháp sắp xếp theo từng bit. Hãy có số danh sách. Trong bước đầu tiên, các phần tử được phân phối trong danh sách tùy thuộc vào phần còn lại mà chúng cho khi chia cho s . Tuy nhiên, yêu cầu chỉ thêm vào cuối danh sách vẫn có hiệu lực. Tiếp theo là nối các danh sách theo thứ tự tăng dần của các phần dư. Quá trình được lặp lại cho đến khi hết các chữ số của các số.

Trong khi sắp xếp theo bit, chúng ta xem xét các chữ số nhị phân của các số, trong phương pháp đã trình bày, chúng ta làm việc với các giá trị của các chữ số trong biểu diễn chữ số s của chúng, tức là chúng ta coi các số như được viết trong hệ thống chữ số s . Do đó tên của thuật toán - một phương pháp của hệ thống số. Đặc biệt, đối với $s = 2$, chúng ta nhận được thuật toán sắp xếp bit.

Về nguyên tắc, không có giới hạn về số lượng danh sách, nhưng phải cẩn thận, vì số lượng danh sách s lớn hơn sẽ làm giảm tốc độ sắp xếp bởi một n đủ nhỏ (ví dụ: $n < s$). Tốt hơn là con số này là lũy thừa của 2 để tìm lượng dư hiệu quả hơn.

Chương trình đính kèm hoạt động với 16 danh sách và số có không quá 8 chữ số thập lục phân. Phần đầu và phần cuối của danh sách được kết hợp thành một mảng 16 phần tử và việc nối của chúng cũng dễ dàng như sắp xếp theo từng bit. Và ở đây, như trong sắp xếp theo bit, chúng ta có thể bỏ việc sử dụng danh sách động, thay thế chúng bằng mảng. Kết nối một lần nữa có thể cực kỳ đơn giản. Với mục đích này, mỗi mảng được cung cấp một con trỏ tới mảng tiếp theo. Tuy nhiên, điều này phải trả giá - cần có thêm bộ nhớ tĩnh.

Chương trình 3.20. Sắp xếp theo hệ số đếm (320radsort.c)

```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define MAX 100

#define BASE 16 /* Cơ sở của hệ thống số */
#define POW2 4 /* 16 = 1 << 4 */
#define DIG_CNT 8 /* Số chữ số */

struct CElem {
    int key;
    /* Các dữ liệu nào đó */
};

struct CList {
    struct CElem data;
    struct CList *next;
};
```

```

};

struct CList *init(unsigned n) /*Điền vào mảng với các số ngẫu nhiê
n*/
{
struct CList *head, *p;
unsigned i;
srand(time(NULL));
for (head = NULL, i = 0; i < n; i++) {
    p = (struct CList *) malloc(sizeof(struct CList));
    p->data.key = rand();
    assert(p->data.key);
    p->next = head;
    head = p;
}
return head;
}

struct CList *radixSort(struct CList *head)
{
struct { struct CList *st, *en; } mod[BASE];
unsigned i, dig, mask, shrM;
/* 1. Khởi tạo */
for (i = 0; i < BASE; i++)
    mod[i].st = (struct CList *) malloc(sizeof(struct CList));

/* 2. Sắp xếp */
mask = BASE-1; shrM = 0;
for (dig = 1; dig <= DIG_CNT; dig++) {

    /* 2.1. Khởi tạo */
    for (i = 0; i < BASE; i++)
        mod[i].en = mod[i].st;

    /* 2.2. Phân phối các phần tử theo danh sách */
    while (NULL != head) {
        /* 2.2.1. Tìm chữ số thứ i trong bản ghi số BASE */
        i = (head->data.key & mask) >> shrM;
        /* 2.2.2. Bao gồm số trong danh sách tương ứng */
        mod[i].en->next = head;
        mod[i].en = mod[i].en->next;
        head = head->next;
    }
}

```

```
 }

/* 2.3. Hợp nhất danh sách*/
mod[BASE-1].en->next = NULL;
for (i = BASE - 1; i > 0; i--)
    mod[i-1].en->next = mod[i].st->next;
head = mod[0].st->next;
/* 2.4. Tính toán mặt nạ mới */
shrM += POW2; mask <= POW2;
}

/* 3. Giải phóng bộ nhớ*/
for (i = 0; i < BASE; i++)
    free(mod[i].st);
return head;
}

void print(struct CList *head)
{ for (; NULL != head; head = head->next)
    printf("%8d", head->data.key);
    printf("\n");
}

void clear(struct CList *head)
{ struct CList *p = head;
    while (NULL != head) {
        head = (p = head) ->next;
        free(p);
    }
}

int main()
{
    struct CList *head;
    head = init(MAX);
    printf("Mảng trước khi sắp xếp:\n");
    print(head);
    head = radixSort(head);
    printf("Mảng sau khi sắp xếp:\n");
    print(head);
    clear(head);
    return 0;
}
```

}

Phương pháp của các hệ thống số có thể được tổng quát hóa hơn nữa cho các số được viết trong bất kỳ hệ thống số nào. Ví dụ, bản ghi có thể được coi là yếu tố cấu thành, mỗi yếu tố được ghi trong hệ thống số riêng của nó. Toàn bộ hồ sơ hóa ra được ghi lại trong một hệ thống số hỗn hợp. Hãy xem cách chúng ta đo thời gian, ví dụ: giây, phút, giờ, ngày, tuần, năm. Đây là một hệ hỗn số với các cơ số $d_1 = 60, d_2 = 60, d_3 = 24, d_4 = 7, d_5 = 52$. Chúng ta có thể xem năm phần tử này như một khóa bản ghi tổng hợp, mỗi phần tử trong số đó được ghi lại trong hệ thống số riêng của nó. Sau đó, trong bước đầu tiên của thuật toán, chúng ta sẽ phân phối các phần tử trong 60 danh sách tùy thuộc vào số giây, trong danh sách thứ hai - một lần nữa trong 60 danh sách được xác định theo số phút, trong danh sách thứ ba - trong 24 danh sách tương ứng với giờ, vân vân.

Bài tập

▷ 3.35. Hãy so sánh sắp xếp theo bit và phương pháp của hệ thống cơ số.

3.3.5. Sắp xếp theo hoán vị

Một đại diện thú vị khác của các phương pháp sắp xếp biến đổi là sắp xếp hoán vị. Sắp xếp hoán vị có thể áp dụng trong các điều kiện hạn chế mạnh hơn so với sắp xếp đếm (xem 3.3.3) hoặc sắp xếp nhiều (xem 3.3.3). Khi sắp xếp theo tập hợp cho hàm thứ tự, chúng ta yêu cầu phải đáp ứng đồng thời hai điều kiện sau (Biểu thị tập hợp của M và số phần tử của nó theo n):

1) các giá trị của f là các số tự nhiên từ một khoảng đóng $[a, b]$ chứa m số nguyên, $m = b - a + 1$. Không phải mọi số nguyên từ $[a, b]$ đều nhất thiết phải là giá trị của f , nhưng tất cả giá trị của f phải từ $[a, b]$.

2) f là vi phân, tức là với $x_1, x_2 \in M, x_1 \neq x_2$ ta có $f(x_1) \neq f(x_2)$ (tức là không lặp lại)

Ở đây chúng ta đặt ra một yêu cầu bổ sung khác:

3) $f : M \rightarrow M$ là xạ ảnh, tức là với mọi $s \in M$ thì tồn tại $x \in M$

sao cho $f(x) = s$.

Gọi số phần tử của M là n , tức là $|M| = n$. Ta ký hiệu S là tập $\{1, 2, \dots, n\}$. Sau đó, ba yêu cầu trên có thể được xây dựng như sau:

$$1) f : S \rightarrow S$$

2) f là một phép song ánh (nghĩa là cùng một lúc là phép đơn ánh và phép tràn ánh)

Trong điều kiện này, hàm bậc f tác dụng lên các phần tử của tập M như một hoán vị. Điều này cung cấp cho chúng ta một thuật toán tuyển tính hiệu quả cho phép chúng ta sắp xếp các phần tử của M tại chỗ mà không cần sử dụng một mảng bổ sung, mà chúng ta cần khi sắp xếp theo nhiều hoặc bằng cách đếm.

Trong quá trình sắp xếp, khóa của các phần tử khác nhau không được so sánh trực tiếp. Kiểm tra duy nhất là liệu phần tử có khóa i có ở vị trí thứ i hay không. Trong trường hợp phần tử có khóa $j, j = m[i], i \neq j$ ở vị trí thứ i , chúng ta sẽ hoán đổi phần tử thứ i và thứ j , trong đó phần tử có khóa j sẽ chuyển sang vị trí thứ j . Nếu sau khi hoán đổi vị trí thứ i có một phần tử có khóa i thì ta chuyển sang xét phần tử thứ $(i + 1)$ tiếp theo của $m[]$. Nếu không, chúng ta thực hiện một trao đổi mới: với phần tử có chỉ số $m[i]$. Để thấy rằng quá trình này là cuối cùng và đến một lúc nào đó phần tử có khóa i sẽ đến vị trí thứ i . (Tại sao?)

Chúng ta sẽ minh họa phương pháp được mô tả trên hoán vị 4375612. Chúng ta bắt đầu với $i = 1$ và hoán đổi liên tiếp phần tử đầu tiên $m[1]$ với $m[m[1]]$ cho đến khi phần tử có khóa 1 chuyển sang vị trí 1 (Chúng ta sẽ chỉ viết các phím, nhấn mạnh các yếu tố mà chúng ta trao đổi.):

4375612
5374612
6374512
1374562

Bây giờ số 1 đã ở đúng vị trí. Chúng ta tiến hành 2:

1374562
1734562
1234567

Số 2 cũng được đặt ra. Nó vẫn còn để đi qua các yếu tố khác và đảm bảo rằng chúng cũng ở đúng vị trí. Quá trình được mô tả rất giống với thuật toán tìm chu trình hoán vị [Nakov-1998]. Trên thực tế, thuật toán lần lượt duyệt qua các chu kỳ hoán vị, với một hoặc hai phần tử sẽ đi đến vị trí cuối cùng của chúng mỗi khi chúng được trao đổi. Rõ ràng là số lượng trao đổi không vượt quá n . Số lần so sánh không vượt quá $2n$. Thật vậy, mỗi phần tử của hoán vị được kiểm tra nhiều nhất hai lần: một lần khi đi qua chu kỳ tương ứng của nó và lần thứ hai khi đi qua vị trí cuối cùng của nó. Trong trường hợp một phần tử được đặt trong hoán vị ban đầu, nó sẽ chỉ tham gia vào một phép so sánh.

Chương trình 3.21. Sắp xếp theo hóa vị(321permsort.c)

```
void permSort(struct CElem m[], unsigned n)
{ unsigned i;
for (i = 0; i < n; i++)
    while (m[i].key != (int)i)
        swap(&m[i], &m[m[i].key]);
}
```

Bài tập

- 3.36. Chứng minh rằng với mọi i ($1 \leq i \leq n$) và với mọi hoán vị của các phần tử của mảng tại trao đổi liên tiếp phần tử $m[i]$ với $m[m[i]]$ tại một điểm $m[i]$ sẽ chứa i .

3.4. Sắp xếp song song

Một tính năng đặc trưng của các thuật toán sắp xếp nổi tiếng hơn là chúng không cho phép thực hiện đồng thời nhiều hơn một phép toán (so sánh). Thoạt nhìn, sự hạn chế như vậy dường như là sự phản ánh hoàn toàn tự nhiên của các đặc điểm kiến trúc của máy tính phổ biến ngày nay. Thật vậy, với một vài trường hợp ngoại lệ, chúng đều là máy nối tiếp trong đó các quá trình tính toán là tuần tự nghiêm ngặt. Tuy nhiên, gần đây, mọi thứ đã bắt đầu thay đổi. Việc cải tiến công nghệ sản xuất, giảm chi phí và liên quan đến việc giảm giá CPU đã tạo nên sự tiến triển ngày càng đáng tin cậy của các hệ thống và cụm đa xử lý. Cho đến gần đây, chỉ kỳ lạ, ngày nay

các hệ thống máy tính với hai hoặc nhiều bộ xử lý bắt đầu được coi là một cái gì đó hoàn toàn bình thường. Sự gia tăng hàng loạt của chúng, kết hợp với sự gia tăng số lượng bộ vi xử lý có thể xảy ra, chắc chắn sẽ dẫn đến sự quan tâm ngày càng tăng đối với các thuật toán cho tính toán song song. Đồng thời, sự cải tiến của công nghệ mạng và giảm tương đối chi phí của hệ thống máy tính và máy chủ sẽ dẫn đến sự công nhận ngày càng tăng của cái gọi là điện toán phân tán, nơi công việc giải quyết một vấn đề được phân chia giữa một số hệ thống máy tính nối mạng. Hơn nữa, ngày nay song song hóa đã được giải quyết ở mức độ chí còn thấp hơn - bên trong chính bộ xử lý. Thật vậy, có quá trình băm bên trong và thực hiện nhiều hoạt động cùng một lúc, với việc thực thi từng hoạt động tiếp theo bắt đầu ngay khi tài nguyên tương ứng trong bộ xử lý được giải phóng, mà không cần đợi quá trình hoàn thành cuối cùng trước đó. Nói chung, ngày nay song song hóa đang tấn công trên mọi mặt trận;))

Mặc dù sự xâm nhập hàng loạt của nó là một hiện tượng tương đối mới, nhưng sẽ là sai lầm nếu coi nó là thành quả của công nghệ hiện đại. Thật vậy, tính song song đã có mặt trong máy tính kể từ khi chúng ra đời. Ví dụ, khái niệm cơ bản của byte liên quan đến việc xử lý đồng thời tám chữ số nhị phân (bit, bit). Song song có thể được tìm thấy trong việc truyền dữ liệu qua bus hoặc mạng, trong hoạt động của các hệ thống nhiều người dùng (máy tính lớn), trong việc thực hiện các phép tính phức tạp trên các hệ thống đa xử lý và hơn thế nữa.

Các ví dụ trên có thể tiếp tục. Tuy nhiên, nó không phải là quá trình song song hóa là quan trọng đối với chúng ta, nhưng tác động của nó đến sự phát triển và sử dụng các thuật toán. Để phát triển khả năng của phần cứng song song, cần có các công cụ phần mềm để điều khiển quá trình phân rã (phân tích) bài toán và phân phối các bài toán con riêng lẻ giữa các thiết bị phần cứng song song (thường là bộ xử lý): cần có các thuật toán song song.

Phương pháp xử lý dữ liệu song song không phải là mới và đã xuất hiện từ lâu trước khi bùng nổ song song đang xuất hiện ngày nay. Trên thực tế, thậm chí rất lâu trước khi thiết bị điện toán đầu tiên xuất hiện. Từ thời xa xưa, mọi người đã nhận thấy rằng nhiều

hoạt động được thực hiện có thể được tăng tốc đáng kể nếu chúng song song với nhau. Ví dụ, trong xây dựng (ví dụ, trong việc xây dựng các kim tự tháp) và hoạt động nông nghiệp, song song hóa dẫn đến gia tốc đáng kể và được sử dụng rộng rãi. Một ví dụ khác về sự song song thành công là cấu trúc của bộ não con người cực kỳ song song: 10 tỷ tế bào thần kinh được kết nối trong một mạng lưới, mỗi tế bào có khoảng 10.000 tế bào khác. Và, mặc dù việc truyền tín hiệu giữa hai tế bào thần kinh được kết nối chậm hơn hàng triệu lần so với các máy tính hiện đại, nhưng có một số tác vụ mà bộ não con người có thể xử lý nhanh hơn nhiều so với máy tính mạnh nhất.

Tất nhiên, không phải mọi quy trình đều có thể được thực hiện song song, và nó thường hóa ra rằng quy trình này phải đi trước quy trình khác. Ví dụ, trước khi chúng ta xây dựng mái của một tòa nhà, chúng ta nên xây dựng mọi thứ khác, bắt đầu từ nền móng.

Có một số thuật toán được sử dụng song song một cách dễ dàng, trong khi những thuật toán khác lại khó hoặc hoàn toàn không. Hơn nữa, không có thuật toán phổ quát cho song song hóa. Mặc dù có một số chương trình chung tạo thuận lợi cho quá trình này, về nguyên tắc, mỗi trường hợp nên được tiếp cận theo cách khác nhau. Vấn đề của song song là cực kỳ phức tạp và nằm ngoài phạm vi của bài báo này.

Dưới đây, chúng ta sẽ tập trung nỗ lực vào việc song song hóa các thuật toán sắp xếp và đặc biệt là phát triển một biến thể sắp xếp song song bằng cách hợp nhất (biến thể không song song cổ điển sẽ được xem xét trong ??). Khi phát triển thuật toán, chúng ta sẽ giới hạn bản thân trong một máy trùu tượng được tạo thành từ nhiều phần tử giống nhau với hai đầu vào và hai đầu ra được đánh số, chúng ta sẽ gọi là máy so sánh. Khi hai số x và y nhập vào đầu vào của một bộ so sánh, đầu ra đầu tiên là $\min(x, y)$ và thứ hai là $\max(x, y)$. Chúng ta sẽ xem xét rằng chúng ta có đủ số lượng bộ so sánh, cũng như bất kỳ số lượng nào trong số chúng có thể hoạt động đồng thời, tức là song song. Vì chúng ta có một thao tác duy nhất được thực hiện bởi trình so sánh, nên số lượng thuật toán sắp xếp mà chúng ta có thể xây dựng sẽ bị giới hạn. Ví dụ, chúng ta sẽ không thể xem xét các thuật toán dựa trên việc đếm hoặc sử dụng các tính năng của biểu diễn nhị phân bên trong của các số và các

thuật toán khác.

Chúng ta sẽ cố gắng tạo sơ đồ của bộ so sánh và đường kết nối (cáp). Chúng ta sẽ muốn tạo một mạch với n đầu vào và n đầu ra, với mỗi đầu vào $\langle a_1, a_2, \dots, a_n \rangle$ cho kết quả là n -bộ $\langle b_1, b_2, \dots, b_n \rangle$, là a hoán vị của các phần tử, nhận được ở đầu vào, tức là có thể nhận được từ đầu ra n -bộ chỉ bằng cách trao đổi các cặp phần tử. Chúng ta muốn a_1, a_2, \dots, a_n là các phần tử của một số tập hợp A trong đó giới thiệu pháp lệnh tuyến tính, tức là mọi phần tử x và y của A đều có thể so sánh được và chính xác một trong ba phần tử là tỷ lệ hợp lệ: $x < y$, $x = y$ hoặc $x > y$ (tính chất nổi tiếng của phép tam phân). Trực quan rõ ràng rằng các phần tử của A có thể được sắp xếp theo thứ tự tăng dần trong một hàng. Dưới đây, để đơn giản, chúng ta sẽ coi các phần tử của A là số. Chúng ta sẽ nhắc lại rằng các tập hợp có thứ tự tuyến tính tạo thành số tự nhiên, số nguyên, số thực, chuỗi ký hiệu, v.v. Hơn nữa, không khó để coi rằng mỗi tập con của một tập hợp có pháp lệnh tuyến tính cũng được sắp xếp theo thứ tự tuyến tính. (Tại sao?)

Chúng ta vẽ các sơ đồ dưới dạng n đường thẳng ngang song song, ở những vị trí được nối thành từng cặp bằng các đường thẳng đứng - các đường so sánh. Chúng ta giả sử rằng sau khi chuyển một cặp dòng qua bộ so sánh, dòng trên sẽ chứa số nhỏ hơn trong số các số được truyền trên hai dòng và dòng dưới - số lớn hơn. Chúng ta giả định rằng thời gian hoạt động của mỗi bộ so sánh là không đổi, ví dụ 1. Trong trường hợp này, mỗi bộ so sánh sẽ chỉ tạo ra một đầu ra sau khi đã nhận được các số trên cả hai dòng đầu vào của nó. Do đó, chỉ một số nhóm bộ so sánh trong mạch sẽ thực sự hoạt động song song, và những nhóm khác sẽ kết thúc hoặc sẽ chờ số ở cả hai đầu vào.

Hãy thử xác định thời gian hoạt động của một mạch so sánh. Giả sử rằng mỗi bộ so sánh thực hiện công việc của nó trong thời gian 1 và việc truyền dữ liệu trên các đường truyền mất thời gian không đáng kể, mà chúng ta có thể coi là 0, chúng ta có thể xác định thời gian hoạt động của mạch là thời gian mà mỗi đường đầu ra nhận được giá trị của nó. Rõ ràng là thời gian hoạt động của mạch trùng với số bộ so sánh tối đa mà qua đó một phần tử của hoán vị đầu vào đi qua trước khi đến đầu ra.

Theo cách tương tự, chúng ta có thể đưa ra định nghĩa đệ quy về độ sâu đường thẳng tại một điểm nhất định. Tất cả các dòng đầu vào có độ sâu bằng 0. Nếu đầu vào của bộ so sánh nhận được các dòng có độ sâu lần lượt là $d(x)$ và $d(y)$, thì các dòng đầu ra của nó có độ sâu $\max(d(x), d(y)) + 1$. Độ sâu của bộ so sánh được định nghĩa là độ sâu của các dòng đi ra khỏi nó và độ sâu của mạch - là độ sâu tối đa của bộ so sánh của mạch. Chúng ta sẽ gọi một sơ đồ sắp xếp nếu với mỗi chuỗi đầu vào $< a_1, a_2, \dots, a_n >$ các phần tử của đầu ra tăng đơn điệu: $b_1 \leq b_2 \leq \dots \leq b_n$.

Trước khi tiếp tục, chúng ta sẽ xây dựng một nguyên tắc đặc biệt hữu ích tạo điều kiện thuận lợi cho việc xác minh các thuật toán và lược đồ sắp xếp.

3.4.1. Nguyên tắc về số không và số một

Định lý 3.1 (Nguyên tắc của số không và số một). *Nếu một thuật toán (lược đồ) sắp xếp chính xác từng chuỗi đầu vào từ 0 và 1, nó cũng sẽ sắp xếp chính xác từng chuỗi đầu vào với các phần tử thuộc một tập hợp có thứ tự tuyến tính.*

Định lý trên còn được gọi là "Nguyên lý của Zeros và Một". Tính hữu ích của nó rõ ràng là rất lớn, vì nó tạo điều kiện thuận lợi đáng kể cho việc chứng minh chính thức (và kiểm tra thực nghiệm) về tính đúng đắn của một thuật toán sắp xếp. Nó đủ để chứng minh rằng nó hoạt động chính xác trên mỗi trong số 2^n dãy số đầu vào 0 và 1 độ dài n , với mỗi n tự nhiên, để chứng minh rằng nó hoạt động chính xác. Nói chung, nguyên tắc này có thể hữu ích khi kiểm tra một chương trình thực hiện thuật toán sắp xếp: Vì số 2^n phát triển tương đối chậm, nhà phát triển có thể dễ dàng kiểm tra tính đúng đắn của chương trình của mình đối với tất cả các chuỗi đầu vào có không quá 30 phần tử. Trước khi tiến hành chứng minh nguyên hàm, chúng ta sẽ hình thành và chứng minh hai bổ đề.

Bổ đề 3.1. Cho ta có một hàm tăng đơn điệu f và một lược đồ với một bộ so sánh duy nhất có đầu vào là $f(x)$ và $f(y)$. Sau đó, tại các đầu ra trên và dưới của nó, lần lượt thu được $f(\min(x, y))$ và $f(\max(x, y))$.

Chứng minh. Thật vậy, từ tính đơn điệu của f mà $\min(f(x), f(y)) =$

$f(\min(x, y))$ và $\max(f(x), f(y)) = f(\max((x, y)))$, trực tiếp theo sau phát biểu của bổ đề. \square

Bổ đề 3.2. Cho là một hàm đơn điệu f . Cho một lược đồ khác của các bộ so sánh được đưa ra, tại đầu vào $a = < a_1, a_2, \dots, a_n >$ cho kết quả là $b = < b_1, b_2, \dots, b_n >$. Khi đó nếu $f(a) = < f(a_1), f(a_2), \dots, f(a_n) >$ được cho ở đầu vào của mạch thì sẽ thu được $f(b) = < f(b_1) ở đầu ra, f(b_2), \dots, f(b_n) >$.

Chứng minh. Ta xét chắc chắn rằng hàm số f đang tăng đơn điệu, tức là với mọi phần tử $x < y$ ta có $f(x) \leq f(y)$. Bây giờ, sử dụng quy nạp dọc theo độ sâu của mỗi dòng, chúng ta có thể chứng minh một phát biểu thậm chí còn chặt chẽ hơn so với bổ đề, từ đó nó sẽ tuân theo trực tiếp, đó là: Nếu một dòng nhận giá trị của y tại đầu vào x , nó sẽ nhận giá trị của $f(y)$ tại đầu vào $f(x)$. Chúng ta sẽ chứng minh phát biểu này bằng cách quy nạp dọc theo độ sâu d của đoạn thẳng:

1) *Cơ sở:* $d = 0$. Với x ở đầu vào ta có y ở đầu ra của dòng. Khi đó, rõ ràng với $f(x)$ ở đầu vào, chúng ta sẽ có $f(y)$ ở đầu ra.

2) *Giả thiết quy nạp:* Giả sử với bất kỳ đường thẳng nào có độ sâu nhỏ hơn d ($d > 0$), nếu ở đầu vào x ta có đầu ra là y , thì ở đầu vào $f(x)$ ta sẽ có đầu ra $f(y)$.

3) *Bước quy nạp:* Cho đầu vào của bộ so sánh có độ sâu d nhận các dòng có giá trị x_1 và x_2 tương ứng. Theo định nghĩa của một bộ so sánh tại hai đầu ra của nó, chúng ta sẽ nhận được $y_1 = \min(x_1, x_2)$ và $y_2 = \max(x_1, x_2)$ tương ứng. Mặt khác, theo giả thiết quy nạp cho hai dòng đầu vào, tại đầu vào $f(x_1), f(x_2)$ trước khi vào bộ so sánh chúng sẽ nhận được các giá trị $f(y_1)$ và $f(y_2)$ tương ứng. Nhưng f là một hàm tăng trưởng đơn điệu. Khi đó, theo Bổ đề 1, tại đầu ra của bộ so sánh, chúng ta sẽ nhận được lần lượt là $\min(f(y_1), f(y_2))$ và $\max(f(y_1), f(y_2))$.

Trường hợp của một hàm giảm đơn điệu f được coi là tương tự. Bổ đề 3.2 được chứng minh. \square

Bây giờ chúng ta đã sẵn sàng để tiến tới việc chứng minh nguyên lý của số không và số một.

Chứng minh (Nguyên tắc của số không và số một): Ở trên trong chứng minh của Bổ đề 3.2 chúng ta đã sử dụng phương pháp quy nạp toán học hoàn chỉnh. Nay giờ chúng ta sẽ sử dụng một phỏng quát khác phương pháp chứng minh nội dung: giả sử ngược lại. Để một lược đồ các bộ so sánh được đưa ra, sắp xếp chính xác tất cả các chuỗi từ 0 đến 1 và giả sử rằng tồn tại một dãy $\langle a_1, a_2, \dots, a_n \rangle$ các phần tử của một số tuyển tính đã sắp xếp thứ tự tập A , mà nó cho một chuỗi không được sắp xếp. Sau đó, nó sẽ tồn tại ít nhất một cặp phần tử a_i và a_j ($a_i < a_j$) sao cho ở đầu ra của mạch a_j nằm ở vị trí nào đó trước a_i . Cho phép bây giờ chúng ta định nghĩa hàm f như sau:

$$f(x) = \begin{cases} 0 & , x \leq a_i; \\ 1 & , x > a_j. \end{cases}$$

Không khó để thấy rằng hàm được định nghĩa như vậy là tăng trưởng đơn điệu. Khi đó từ thực tế rằng a_j đứng trước a_i bởi Bổ đề 3.2, chúng ta nhận được rằng $f(a_j)$ đứng trước $f(a_i)$. Mặt khác, theo định nghĩa của f , chúng ta có $f(a_j) = 1$ và $f(a_i) = 0$. Nhưng khi đó lược đồ đã xét không sắp xếp đúng dãy $\langle f(a_1), f(a_2), \dots, f(a_n) \rangle$, là một dãy 0 và 1. Chúng ta có một mâu thuẫn với giả thiết rằng lược đồ sắp xếp đúng tất cả các dãy 0 và 1.

3.4.2. Trình tự bitonic

Nguyên tắc vừa được chứng minh sẽ được sử dụng cụ thể đối với chúng ta trong phát biểu sau đây. Dưới đây chúng ta sẽ cố gắng phát triển một thuật toán sắp xếp song song hiệu quả dựa trên các mạch so sánh. Để làm điều này, trước tiên chúng ta sẽ xem xét một thuật toán để sắp xếp các chuỗi bitonic. Theo chuỗi bitonic, chúng ta có nghĩa là một chuỗi tăng trưởng đơn điệu, sau đó giảm đơn điệu, hoặc ngược lại: giảm đơn điệu, sau đó tăng đơn điệu. Ví dụ:

$\langle 1, 7, 9, 11, 27, 5, 4, 3, 3, 1 \rangle$

$\langle 37, 20, 17, 16, 10, 10, 10, 18, 25, 47 \rangle$

Đặc biệt, mọi thứ được sắp xếp theo thứ tự tăng dần hoặc giảm dần cũng là bitonic. Các chuỗi bitonic bao gồm 0 và 1 có dạng $0^i 1^j 0^k$ hoặc $1^i 0^j 1^k$, $i, j, k \geq 0$. Tận dụng nguyên tắc của số không và số một,

dưới đây chúng ta sẽ giới hạn trong việc phát triển - của lược đồ sắp xếp trình tự biton chỉ từ 0 và 1.

3.4.3. "Rõ ràng một nửa"

Để đơn giản hóa việc lập luận dưới đây, chúng ta sẽ làm việc với các dãy có một số phần tử chẵn. Theo các giả định này, chúng ta sẽ phát triển một lược đồ các bộ so sánh, hoạt động theo nghĩa tương tự như sắp xếp nhanh (xem 3.2.6), và chúng ta sẽ gọi là "rõ ràng một nửa". Khi nhận được một chuỗi bitonic có độ dài n ở đầu vào của đầu ra, sẽ thu được hai chuỗi bitonic, mỗi chuỗi có độ dài $n/2$. Trong trường hợp này, các giá trị lớn hơn sẽ nằm trong dãy thứ hai, tức là mỗi phần tử của dãy thứ nhất sẽ nhỏ hơn hoặc bằng mỗi phần tử của dãy thứ hai. Từ đó rõ ràng là có ít nhất một trong hai chuỗi chỉ bao gồm 0 (trên) hoặc chỉ 1 (dưới), do đó có tên "rõ ràng một nửa". Lược đồ được xem xét có độ sâu 1 và thực hiện song song tất cả các phép so sánh thuộc loại $(i, n/2 + i)$ cho $i = 1, 2, \dots, n/2$.

3.4.4. Sắp xếp chuỗi bitonic

Giả sử rằng $n/2$ cũng là số chẵn. Sau đó, chúng ta có thể áp dụng "một nửa rõ ràng" cho mỗi trong hai trình tự mới thu được. Nếu $n/4$ lại chẵn, thì chúng ta có thể tiếp tục quá trình trên từng trong bốn trình tự mới, v.v. Nếu n là lũy thừa của 2, quá trình được mô tả có thể được tiếp tục một cách đệ quy cho đến khi đạt đến các chuỗi có độ dài 1. Không khó để thấy rằng bất kỳ chuỗi bitonic nào cũng có thể được sắp xếp theo cách này. (Tại sao?)

Chúng ta hãy thử đánh giá mức độ phức tạp của quy trình được mô tả. Vì mỗi bước yêu cầu thời gian 1 (thời gian để chạy "xóa một nửa"), bộ so sánh hoạt động song song (theo định nghĩa của lược đồ bộ so sánh) và độ dài của các hàng được xem xét giảm đi một nửa mỗi lần, tổng thời gian cần thiết là $\log_2 n$. Chúng ta đã thu được một thuật toán để *sắp xếp các chuỗi bitonic 0 và 1* với độ phức tạp $\Theta(\log_2 n)$. Nay giờ nó tuân theo nguyên tắc của các số không và các số không mà lược đồ đề xuất hoạt động cho một chuỗi bitonic tùy ý, tức là không chỉ từ 0 và 1.

3.4.5. Sắp xếp sơ đồ hợp nhất

Bây giờ chúng ta có một thuật toán song song hiệu quả để sắp xếp các chuỗi bitonic. Chúng ta sẽ cố gắng tóm tắt nó để chúng ta có được một thuật toán hiệu quả để sắp xếp các chuỗi ngẫu nhiên, mà không nhất thiết yêu cầu chúng phải bitonic. Để làm điều này, trước tiên chúng ta sẽ phát triển một thuật toán để hợp nhất hai chuỗi đã được sắp xếp thành một chuỗi được sắp xếp chung như được mô tả bởi Batcher.

Nếu chúng ta đảo ngược trình tự thứ hai, chúng ta sẽ nhận được một trình tự bitonic mà chúng ta đã biết cách sắp xếp. Rõ ràng là quá trình nghịch đảo có thể diễn ra trong thời gian 1 khi sử dụng đủ số lượng bộ so sánh. Sau đó, chúng ta có thể áp dụng sơ đồ sắp xếp theo trình tự bitonic đã phát triển ở trên. Tổng thời gian cần thiết để hợp nhất hai chuỗi sẽ là $1 + \log_2 n$. Nếu chúng ta xem xét kỹ hơn hai bước đầu tiên của thuật toán kết quả, chúng ta có thể thấy rằng chúng có thể được kết hợp tương đối dễ dàng thành một, tức là, dãy thứ hai có thể được đảo ngược đồng thời và "xóa một nửa". Với mục đích này, thay vì so sánh các cặp $(i, n/2 + i)$, chúng ta nên so sánh $(i, n - i + 1)$, với $i = 1, 2, \dots, n/2$. Kết quả là, chúng ta nhận được hai chuỗi bitonic, với mỗi phần tử của phần tử thứ nhất nhỏ hơn hoặc bằng mỗi phần tử của phần thứ hai. Sau đó, chúng ta có thể tiếp tục áp dụng phân loại bitonic trực tiếp từ bước thứ hai. Kết quả là, chúng ta nhận được một thuật toán để hợp nhất hai chuỗi đã được sắp xếp cho nhặt kí thời gian $\log_2 n$.

3.4.6. Sắp xếp sơ đồ phân loại

Khi chúng ta đã triển khai một lược đồ hiệu quả của các bộ so sánh để hợp nhất các chuỗi đã sắp xếp, giờ đây chúng ta đã sẵn sàng để tiến tới mục tiêu thực sự của mình: phát triển một thuật toán hiệu quả để sắp xếp bất kỳ chuỗi đâu vào nò. Chúng ta sẽ tập trung nỗ lực vào việc thực hiện phiên bản sắp xếp hợp nhất song song (xem ??) Dựa trên đề án sáp nhập ở trên.

Để tạo điều kiện thuận lợi cho việc suy luận thêm, chúng ta hãy giả sử rằng n là một lũy thừa của 2. Chúng ta sẽ suy nghĩ theo cách quy nạp. Giả sử chúng ta có hai dãy đã được sắp xếp, mỗi dãy có độ

dài $n/2$. Sau đó, chúng ta có thể hợp nhất chúng thành một chuỗi được sắp xếp chung cho nhặt ký thời gian $\log_2 n$ bằng cách sử dụng lược đồ hợp nhất. Nhưng làm thế nào để chúng ta có được hai chuỗi được sắp xếp? Giả sử chúng ta có 4 dây đã sắp xếp l_1, l_2, l_3, l_4 với độ dài $n/4$ mỗi dây. Sử dụng lược đồ hợp nhất, chúng ta có thể hợp nhất l_1 và l_2 trong một chuỗi được sắp xếp chung có độ dài $n/2$ cho nhặt ký thời gian $2(n/2)$. Chúng ta có thể hợp nhất l_3 và l_4 cùng một lúc, sử dụng bản sao thứ hai của sơ đồ sáp nhập mà không cần thêm thời gian. Kết quả là, chúng ta sẽ nhận được hai dây đã được sắp xếp theo yêu cầu có độ dài $n/2$. Và làm thế nào để có các dây số l_1, l_2, l_3, l_4 ? Mọi thứ cũng tương tự như vậy. Một lần nữa, chúng ta giả sử rằng chúng ta có tám chuỗi đã được sắp xếp với độ dài $n/8$, chúng ta tập hợp và hợp nhất thành từng cặp cho nhặt ký thời gian $2(n/8)$, v.v. Chúng ta tiếp tục giảm kích thước của bài toán một cách đệ quy để thu được các chuỗi đơn phần tử, mỗi chuỗi chúng ta có thể xem xét được sắp xếp. Nay giờ, ngược lại của đệ quy, chúng ta nhận được một chuỗi đã được sắp xếp.

Hãy thử ước tính độ sâu của sơ đồ phân loại được xây dựng theo cách này. Độ sâu $D(n)$ của mạch sắp xếp thứ tự n phần tử có thể được tính bằng độ sâu $D(n/2)$ của mạch sắp xếp thứ tự $(n/2)$ (trên thực tế, chúng ta có hai lược đồ như vậy, nhưng chúng hoạt động song song), cộng với độ sâu $\log_2 n$ của mạch hợp nhất. Chúng ta thu được sự phụ thuộc lặp lại sau:

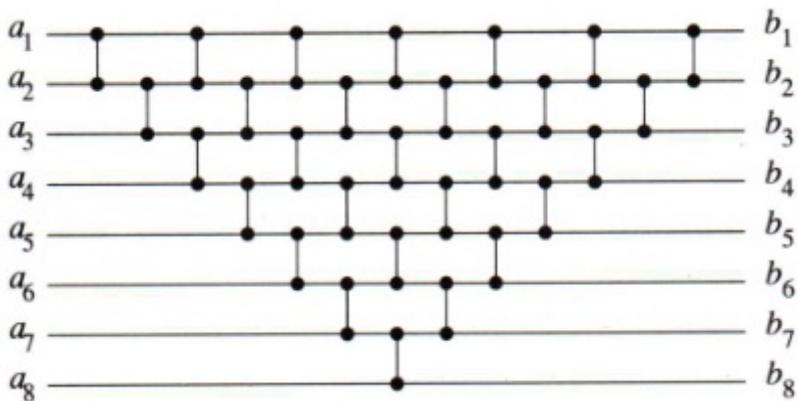
$$D(n) = \begin{cases} 0 & n = 1; \\ D\left(\frac{n}{2}\right) + \log_2 n & n = 2^k, k \geq 1. \end{cases}$$

Từ đây theo Định lý Cơ bản (xem ??) Chúng ta nhận được:

$$D(n) \in \Theta\left(\log_2^2 n\right).$$

3.4.7. Sơ đồ phân loại chuyển vị

Tất nhiên, phương án đề xuất ở trên không phải là phương án phân loại khả thi duy nhất. Hầu như mọi thuật toán sắp xếp phổ biến chỉ trao đổi các phần tử tạo ra một lược đồ sắp xếp tương ứng bằng các bộ so sánh. Hình 3.3 hiển thị sơ đồ tương ứng với việc sắp xếp theo cách chèn (xem 3.2.2).



Hình 3.3. Lược đồ sắp xếp để phân loại chèn.

Bởi vì phân loại chèn chỉ so sánh các phần tử liền kề, các bộ so sánh trong sơ đồ trên chỉ kết nối các dòng liền kề. Tình hình cũng tương tự với phương pháp bong bóng. Các lược đồ trong đó các bộ so sánh chỉ kết nối các cặp đường liền nhau được gọi là các *lược đồ chuyển vị*. Tất nhiên, không phải tất cả các lược đồ chuyển vị đều là bộ sắp xếp.

3.4.8. Sơ đồ sáp nhập Butcher chẵn lẻ

Sơ đồ hợp nhất chẵn-lẻ được Butcher phát triển vào năm 1960. Ý tưởng như sau: Hãy cho hai dãy n phần tử được sắp xếp $< a_1, a_2, \dots, a_n >$ và $< a_{n+1}, a_{n+2}, \dots, a_{2n} >$, mà chúng ta muốn hợp nhất thành một dãy được sắp xếp chung (n là lũy thừa của 2). Chúng ta sẽ suy nghĩ một cách linh hoạt. Hãy để chúng ta hợp nhất số lẻ ($< a_1, a_3, \dots, a_{n-1} >$ và $< a_{n+1}, a_{n+3}, \dots, a_{2n-1} >$) và chẵn ($< a_2, a_4, \dots, a_n >$ và $< a_{n+2}, a_{n+4}, \dots, a_{2n} >$) phần tử của cả hai dãy. Bây giờ chúng ta có thể kết hợp các chuỗi chẵn và lẻ, sử dụng bộ so sánh $n/2$, được đặt giữa $2i - 1$ và $2i$ cho $i = 1, 2, \dots, n$. Kết quả là, chúng ta thu được một lược đồ có độ sâu $\Theta(\log_2 n)$.

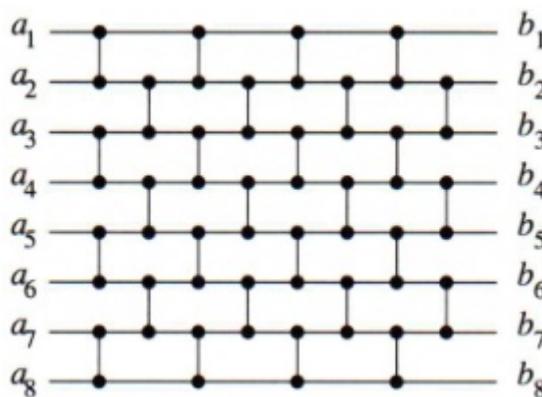
3.4.9. Lược đồ sắp xếp chẵn-lẻ

Phát triển thêm ý tưởng về sơ đồ hợp nhất chẵn-lẻ, Butcher đã quản lý để có được một sơ đồ sắp xếp với độ sâu $\Theta(\log_2^2 n)$. Ở đây

các bộ so sánh ở mức n và vị trí của chúng được tính bằng một công thức đơn giản. Chúng ta hãy biểu thị bằng i là số của dòng ($i = 2, 3, \dots, n - 1$) và bằng d - độ sâu của bộ so sánh ($d = 1, 2, \dots, n$). Khi đó dòng i nối với dòng $j = i + (-1)^{i+d}, 1 \leq j \leq n$ ở độ sâu d . Đó là, lược đồ có thể được xây dựng theo thuật toán sau: (xem Hình 3.4)

Với $d = 1, 2, \dots, n$ chúng ta lặp lại:

- 1) Nếu d chẵn, nối các dòng $2i - 1$ và $2i$ với một bộ so sánh, với $i = 1, 2, \dots, [n/2]$.
- 2) Nếu d là số lẻ, nối các dòng $2i$ và $2i + 1$ bằng một bộ so sánh, với $i = 1, 2, \dots, [(n - 1)/2]$.



Hình 3.4. Lược đồ sắp xếp chẵn-lẻ.

3.4.10. Lược đồ hoán vị

Hãy xem xét một mạch trong đó công tắc hai trạng thái được sử dụng thay vì bộ so sánh, có thể bật và tắt. Giống như bộ so sánh, bộ chuyển mạch cũng có hai đầu vào và hai đầu ra. Khi đầu ra công tắc tắt, các giá trị nhận được ở đầu vào được cấp không thay đổi và khi bật, chúng được trao đổi (gạch chéo). Giả sử là một lược đồ có n dòng và đưa ra giá trị đầu vào của nó là $<1, 2, \dots, n>$. Rõ ràng rằng, bằng cách điều chỉnh các công tắc, chúng ta có thể thu được các hoán vị khác nhau của n số tự nhiên đầu tiên. Trong trường hợp chúng ta có thể nhận được tất cả $n!$ hoán vị, chúng ta sẽ gọi là hoán

vị lược đồ. Có thể chỉ ra rằng việc thay thế tất cả các bộ so sánh bằng các công tắc trong bất kỳ sơ đồ sắp xếp nào sẽ dẫn đến một lược đồ hoán vị.

Hầu hết các nguồn đều chỉ ra Armstrong, Nelson và O'Connor là những nhà nghiên cứu đầu tiên về kế hoạch sắp xếp công bố nghiên cứu của họ vào năm 1954. Sau đó vào năm 1960, Butcher đã phát triển kế hoạch sắp xếp và hợp nhất chẵn lẻ của mình. Chúng ta cũng nợ phân loại dựa trên chuỗi bitonic. Trong một thời gian dài, đây là những thuật toán được biết đến nhiều nhất - với độ phức tạp của thuật toán $\Theta(\log_2^2 n)$.

Năm 1983, Aitai, Komlos và Zemeredi đã cải thiện đáng kể kết quả này bằng cách phát triển một sơ đồ sắp xếp với độ sâu $\Theta(\log_2 n)$ sử dụng bộ so sánh $\Theta(\log_2 n)$. Mặc dù có độ phức tạp theo lôgarit, nhưng thuật toán này rất khó áp dụng trong thực tế do hằng số rất lớn紧跟 sau $\Theta(\dots)$.

Bài tập

- ▷ 3.37. Chứng minh rằng thuật toán của 3.4.4 sắp xếp chính xác từng chuỗi bitonic (xem 3.4.4).
- ▷ 3.38. Chứng minh rằng mỗi lược đồ chuyển vị sắp xếp với n đầu vào chứa $\Omega(n^2)$ bộ so sánh (xem 3.4.7).
- ▷ 3.39. Xác định độ sâu và tìm số bộ so sánh trong sơ đồ chuyển vị với n đầu vào (xem 3.4.7), Tương ứng với:
 - (a) phân loại chèn (xem 3.2.2);
 - (b) phân loại bong bóng (xem 3.2.4).
- ▷ 3.40. Sử dụng quy nạp theo số dòng, chứng minh rằng một lược đồ chuyển vị có n dòng được sắp xếp nếu và chỉ khi nó sắp xếp theo dãy $(n, n-1, \dots, 1)$.
- ▷ 3.41. Chứng minh rằng với $n > 2$ đối với mỗi mạch hoán vị có n đầu vào sẽ có ít nhất một hoán vị, mà có ít nhất hai cách khác nhau (xem 3.4.10).

3.5. Câu hỏi và bài tập

▷ 3.42. So sánh các phương pháp sắp xếp sơ cấp.

Để so sánh trong thực tế các phương pháp sắp xếp cơ bản: chèn (xem 3.2.2), Bong bóng (xem 3.2.4) Hoặc lựa chọn trực tiếp (xem 3.2.8). Để thực hiện lý luận lý thuyết và kiểm tra thực nghiệm cho các vấn đề khác nhau:

(a) số phần tử: 10; 20; 50; 100; 1000; 10000;

b) thứ tự của mảng: có thứ tự; sắp xếp lại; khuấy nhẹ; bị khuấy động mạnh.

▷ 3.43. Đủ số lần lặp.

Chứng minh rằng đối với nội tiếp, phép chọn trực tiếp, bong bóng và hình chóp $n - 1$ lần lặp của trường hợp ngoại tiếp là đủ.

▷ 3.44. So sánh các phương pháp sắp xếp nhanh.

Để so sánh giữa phân loại nhanh theo lý thuyết và thực nghiệm (xem 3.2.6), Phương pháp phân loại thỏ và rùa (xem 3.2.7) Và phân loại theo hình chóp (xem 3.2.9).

▷ 3.45. Kết hợp phân loại nhanh với một phương pháp sơ cấp.

Để xác định phương pháp sắp xếp cơ bản nào (chèn, bong bóng hoặc lựa chọn trực tiếp: xem 3.2.2, 3.2.4 và 3.2.8) Về mặt lý thuyết, tốt nhất là kết hợp với sắp xếp nhanh (xem 3.2.6). Biên dịch việc triển khai chương trình thích hợp và thực hiện các bài kiểm tra thích hợp. Các giả thiết lý thuyết sơ bộ đã được xác nhận trong thực tế chưa?

▷ 3.46. Trường hợp xấu nhất và tốt nhất.

Đối với mỗi thuật toán sắp xếp bằng cách so sánh để xác định trường hợp xấu nhất và tốt nhất của nó.

▷ 3.47. Sự bền vững.

Thuật toán sắp xếp nào được thảo luận trong chương này là mạnh mẽ?

▷ 3.48. Truy cập nhất quán.

Thuật toán nào được coi là có thể áp dụng để sắp xếp tệp và danh sách tuyến tính mà không có khả năng truy cập trực tiếp vào các phần tử?

► 3.49. Đảm bảo số lượng so sánh và trao đổi tối thiểu.

Thuật toán sắp xếp nào được thảo luận trong chương này đảm bảo số lượng tối thiểu trong trường hợp xấu nhất:

- a) so sánh;
- b) trao đổi.

► 3.50. Số lần trao đổi tối thiểu.

Một mảng các số nguyên không có các phần tử bằng nhau và quan hệ giữa một số phần tử của nó được đưa ra. Để biên dịch một thuật toán sắp xếp mảng, thực hiện một số lượng trao đổi tối thiểu.

► 3.51. Phân loại kép.

Một ma trận $A = (a_{ij})$ được đưa ra. Mỗi hàng của nó được sắp xếp riêng lẻ theo thứ tự tăng dần, sau đó các trụ của nó được sắp xếp riêng biệt. Các hàng có được sắp xếp theo thứ tự tăng dần không?

CHƯƠNG 4

TÌM KIẾM VÀ THUẬT TOÁN

4.1. Giới thiệu	303
4.2. Tìm kiếm liên tiếp	306
4.2.1. Tìm kiếm liên tiếp trong danh sách đã sắp xếp	309
4.2.2. Tìm kiếm liên tục với sự sắp xếp lại	311
4.3. Tìm kiếm theo từng bước. Tìm kiếm bậc hai	313
4.4. Tìm kiếm nhị phân	317
4.5. Tìm kiếm Fibonacci	323
4.6. Tìm kiếm nội suy	326
4.7. Câu hỏi và bài tập	328

4.1. Giới thiệu

Quá trình sắp xếp được thảo luận trong Chương 3 có liên quan chặt chẽ đến quá trình tìm kiếm. Thật vậy, việc sắp xếp một phần hoặc toàn bộ một tập hợp các mục có thể đẩy nhanh quá trình tìm kiếm. Tất nhiên, hai hoạt động cũng có thể được coi là hoàn toàn độc lập. Ví dụ, tên của nhân viên trong một công ty có thể được sắp xếp theo thứ tự bảng chữ cái, chỉ để có một danh sách được in ra giấy. Đồng thời, hầu hết các thuật toán tìm kiếm không giả định hoặc yêu cầu sắp xếp trước.

Tìm kiếm là một hoạt động cơ bản được thực hiện hàng ngày bởi mỗi chúng ta vào nhiều dịp khác nhau. Cho dù chúng ta có nhận ra hay không, thì cứ như thế chúng ta dành một khoảng thời gian đáng kể để tìm kiếm: tìm số điện thoại, tìm chìa khóa bị mất, tìm thứ gì đó để ăn trong tủ lạnh, tìm Zamunda trên bản đồ Châu Phi, tìm kiếm làm việc, tìm kiếm một chương trình truyền hình thú vị, chúng ta đang tìm kiếm vấn đề mới của "Lao động vàng", chúng ta đang tìm kiếm sự cố, v.v. Có thể nói rằng nhu cầu là một hoạt động nổi tiếng đối với tất cả chúng ta. Tuy nhiên, chỉ có một số người đã nghĩ về

các phương pháp được sử dụng. Đồng thời, sự đa dạng của các loại hình tìm kiếm gần như là vô hạn nên rất khó phân loại chúng. Trong quá trình tìm kiếm, hầu hết chúng ta tiến hành chủ yếu từ kinh nghiệm tích lũy được, và trong một tình huống không quen thuộc, họ xử lý theo phương pháp thử và sai. Có những phương pháp tìm kiếm đủ mạnh có thể khác biệt đáng kể với nhau tùy thuộc vào các điều kiện cụ thể. Ví dụ: tìm kiếm có thể được đặt không chính xác (tìm kiếm một con bê dưới một con bò), mục tiêu có thể được xác định mơ hồ (tìm kiếm một chương trình truyền hình thú vị), mục tiêu có thể di chuyển (tìm kiếm tàu ngầm của đối phương), tìm kiếm nó có thể bị giới hạn về thời gian (tìm bom bằng cơ chế đồng hồ), các mẫu có thể có giá khác (mỏ kim đáy bể), tập hợp có thể là vô tận (tìm kiếm dầu) và các loại khác. Đặc biệt, tìm kiếm có thể có tất cả các thuộc tính được liệt kê ở trên, cũng như có thể có một số thuộc tính khác (tìm kiếm ý nghĩa cuộc sống).

Rõ ràng là hầu như không thể bao gồm tất cả các trường hợp được liệt kê ở trên, đó là lý do tại sao một số trong số chúng cần được sửa chữa. Dưới đây, chúng ta sẽ làm việc với một mô hình đơn giản và được xác định rõ với các thuộc tính sau:

- mục tiêu cố định và được xác định rõ ràng;
- thời gian là không giới hạn;
- tập mà chúng ta tìm kiếm là hữu hạn và tĩnh, tức là nó không thay đổi trong quá trình tìm kiếm;
- tất cả các mẫu (so sánh) có cùng một mức giá;
- chúng ta không bao giờ mắc lỗi trong bài kiểm tra;
- tất cả thông tin từ các so sánh trước đó được giữ lại.

Tìm kiếm hiếm khi là một hoạt động cô lập. Mô hình trên rất hữu ích, nhưng không may là không phải lúc nào cũng có sẵn. Thường xảy ra rằng tập hợp mà bạn đang xem thay đổi động: các phần tử được bao gồm và loại trừ, sắp xếp lại, v.v. Do đó, sẽ rất hữu ích nếu coi các thuật toán tìm kiếm không phải là một cái gì đó riêng biệt và độc lập, mà là một phần của một gói hoàn chỉnh các phép toán cơ bản trên các phần tử của một tập hợp. Bằng cách này, khi xem xét tất cả các hoạt động cùng nhau, chúng ta sẽ có thể đánh giá tốt

hơn sự tương tác của chúng và tránh các ước tính không chính xác về hiệu suất tìm kiếm.

Ví dụ, tìm kiếm nhị phân, sẽ được thảo luận dưới đây, có độ phức tạp là $\Theta(\log_2 n)$ trong cả trường hợp trung bình và trường hợp xấu nhất. Tuy nhiên, đồng thời, nó yêu cầu các phần tử của tập hợp phải được sắp xếp. Nếu không đúng như vậy, việc sắp xếp trước chúng trước mỗi lần tìm kiếm sẽ yêu cầu thời gian theo thứ tự là $\Theta(n \cdot \log_2 n)$. Do đó, độ phức tạp thực tế của nó sẽ là $\Theta(n \cdot \log_2 n)$. Tất nhiên, chúng ta có thể giữ cho đám đông luôn được sắp xếp. Tuy nhiên, điều này sẽ làm phức tạp đáng kể các hoạt động bao gồm và loại trừ các phần tử. Thật tiện lợi khi coi tìm kiếm như một phần tử của tập hợp các thao tác sau:

- khởi tạo
- tìm kiếm
- chèn
- xóa
- thống nhất các bộ
- sắp xếp

Có một kết nối trực tiếp giữa một số hoạt động trên và chúng thường được thực hiện đồng thời. Ví dụ, trước khi chèn một mục, bạn thường tìm kiếm vị trí tương ứng. Như với việc sắp xếp (và các cấu trúc dữ liệu được thảo luận trong Chương 2), chúng ta sẽ giả định rằng các phần tử của tập hợp được đề cập là các bản ghi kiểu struct CElem, một trong các trường được chọn làm khóa. Đôi khi các trường khóa là duy nhất trong các phần tử của tập hợp, những trường hợp khác lặp lại chúng có thể chấp nhận được. Dưới đây chúng ta sẽ thấy rằng khả năng lặp lại hoặc không lặp lại các phần tử của tập hợp hóa ra lại là một yếu tố quan trọng của việc thực hiện tập hợp các phép toán trên. Giả sử rằng đối với các hoạt động trên, chúng ta đã triển khai các thuật toán thích hợp để ngăn chặn sự trùng lặp của các trường khóa một cách rõ ràng. Chúng ta có thể dễ dàng xử lý mã chương trình liên quan để cho phép sao chép không? Một cách tiếp cận khả thi để giải quyết vấn đề là thêm một con trỏ bổ sung vào bản ghi đại diện cho các phần tử của tập

hợp. Con trỏ này sẽ trỏ đến danh sách các mục có cùng khóa. Ưu điểm chính của phương pháp này là chỉ cần một lần tìm kiếm là đủ để tìm tất cả các khóa có giá trị cho trước. Một cách triển khai khá thi khác là cho phép sao chép các phần tử và khi tìm kiếm, bất kỳ phần tử nào cũng được trả về với khóa được chỉ định. Vấn đề chính ở đây sẽ là tìm kiếm tất cả các phần tử với khóa này, mà nếu cần, một cơ chế đặc biệt nên được cung cấp. Tất nhiên, chúng ta có thể giới thiệu một khóa kinh doanh thứ hai, mà chúng ta có thể đảm bảo là duy nhất cho toàn bộ. Sau đó, chúng ta chỉ có thể tìm kiếm khóa đầu tiên và cả hai khóa cùng một lúc. Có lẽ tùy chọn linh hoạt nhất là triển khai một hàm so sánh đặc biệt mà thủ tục tìm kiếm có thể tham khảo. Bằng cách này, trong quá trình tìm kiếm sẽ có thể kiểm tra xem mục hiện đang được xem xét với khóa được tìm kiếm có đáp ứng bất kỳ bộ điều kiện bổ sung nào hay không.

4.2. Tìm kiếm liên tiếp

Thuật toán tìm kiếm cơ bản nhất và rõ ràng nhất là tìm kiếm nhất quán. Giả sử rằng các phần tử của tập hợp được chứa trong mảng một chiều. Việc tìm kiếm được thực hiện bằng cách tìm kiếm tuần tự các phần tử của mảng cho đến khi tìm thấy phần tử cần tìm hoặc cho đến khi tìm thấy tất cả các phần tử. Điều sau có nghĩa là một phần tử có khóa như vậy không tồn tại. Khi một phần tử mới đến, chúng ta sẽ chèn nó vào cuối mảng seqSearch(). Sau đây là cách triển khai có thể có của các chức năng cơ bản dựa trên tìm kiếm nhất quán:

Chương trình 4.1. Tìm kiếm liên tiếp (401seq-arr.c)

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
#define DataType int

struct CElem {
    int key;
    DataType data;
    /* ... */
}
```

```

} m[MAX + 1]; /* Một mảng các bản ghi */
unsigned n; /* Số phần tử trong mảng */

void seqInit(void) { n = 0; } /* Khởi tạo*/
unsigned seqSearch(int key) /*Tìm kiếm liên tiếp */
{ unsigned x;
  m[0].key = key; /*Giới hạn*/
  for (x = n + 1; key != m[--x].key; ) ;
  return x;
}

void seqInsert(int key, DataType data) /*Thêm một phần tử mới*/
{ m[++n].key = key;
  m[n].data = data;
}

void seqPrint(void) /*Hiển thị danh sách trên màn hình*/
{ unsigned i;
  char buf[9];
  for (i = 1; i <= n; i++) {
    sprintf(buf, "%d | %d", m[i].key, m[i].data);
    printf("%8s", buf);
  }
}

void performSearchTest(void)
{ unsigned ind, elem2Search;
  for (elem2Search = 0; elem2Search < 2*MAX; elem2Search++) {
    printf("Đang tìm kiếm một phần tử có khóa %u.\n", elem2Search);
    if (0 == (ind = seqSearch(elem2Search)))
      printf("%s", "Một phần tử có khóa như vậy không tồn tại!\n");
    else
      printf("%sĐã tìm thấy phần tử! phần thông tin: %d\n", m[ind].data);
  }
}

int main()
{ unsigned ind;
  seqInit();

```

```

for (ind = 0; ind < MAX; ind++)
    seqInsert(rand() % (MAX*2), ind);
printf("Danh sách bao gồm các mục sau: \n");
seqPrint();
printf("\n Thủ nghiệm:\n");
performSearchTest();
return 0;
}

```

Như với việc sắp xếp, chúng ta đã giả định ở trên rằng phần thông tin thực tế của các phần tử mảng chỉ chứa trường dữ liệu. Tất nhiên, nếu cần, người đọc có thể dễ dàng sửa đổi cấu trúc của loại **struct** CElem theo nhu cầu và yêu cầu cụ thể của họ, điều này sẽ yêu cầu những thay đổi tối thiểu tương ứng trong đoạn mã trên.

Việc tìm kiếm được thực hiện tuần tự theo hướng giảm dần các chỉ số. Lưu ý rằng phần tử rỗng của mảng không chứa phần tử của tập hợp mà là thông tin dịch vụ. Nó được sử dụng như một phần tử giới hạn cho phép đơn giản hóa chương trình: chỉ một điều kiện được kiểm tra, vì chúng ta đã đảm bảo tìm được phần tử có khóa được chỉ định (trong trường hợp cực đoan là số 0). Trong trường hợp có khóa trùng lặp, thuật toán trên hiển nhiên trả về phần tử nhận được cuối cùng với khóa này.

Độ phức tạp của thuật toán ở giữa và trong trường hợp xấu nhất tương ứng là bao nhiêu? Rõ ràng rằng trường hợp xấu nhất là khi thiếu vật phẩm cần thiết. Sau đó, tất cả $n + 1$ phần tử của mảng được xem, bao gồm cả phần tử rỗng. Trong trường hợp chung, khi tìm kiếm thành công, số lần so sánh trung bình là $(n + 1)/2$ (Tại sao?). Đồng thời, cần quét toàn bộ mảng để tìm tất cả các phần tử có khóa cho trước.

Bài tập

- ▷ 4.1. Chứng minh rằng số phép so sánh trung bình trong một tìm kiếm tuần tự là $(n + 1)/2$.

4.2.1. Tìm kiếm liên tiếp trong danh sách đã sắp xếp

Tìm kiếm tuần tự, đã thảo luận ở trên, là thuật toán tìm kiếm đơn giản nhất và không hiệu quả nhất. Chúng ta có thể cố gắng cải thiện hiệu quả của nó bằng cách sắp xếp các mục. Điều này sẽ tương đối dễ dàng nếu chúng ta sử dụng danh sách liên kết thay vì một mảng. Tất cả những gì chúng ta phải làm là chèn từng mục mới vào đúng vị trí của nó để danh sách vẫn được sắp xếp. Do đó, trước mỗi lần chèn sẽ phải thực hiện một tìm kiếm tương ứng để tìm ra vị trí tương ứng.

Thao tác chèn phức tạp đáng kể. Chúng ta đã làm điều đó ở trên trong một thời gian liên tục, không có bất kỳ so sánh nào và bây giờ nó sẽ yêu cầu tìm kiếm sơ bộ. Rốt cuộc chúng ta đã đạt được những gì? Thực tế là các mục được sắp xếp theo thứ tự tăng dần sẽ cho phép chúng ta ngừng tìm kiếm thêm khi đến một mục có khóa lớn hơn tìm kiếm. Số lượng phép so sánh cần thiết trong trường hợp tìm kiếm không thành công giảm xuống $(n + 1)/2$, so với $n + 1$ trong trường hợp không được sắp xếp. Trên thực tế, các tìm kiếm thành công và không thành công trở nên ngang nhau, vì việc sửa đổi này rõ ràng không ảnh hưởng đến tìm kiếm thành công. Trong trường hợp xấu nhất, tất cả các yếu tố phải được xem xét lại, tức là $n + 1$ phép so sánh phải được thực hiện. Hóa ra là để đánh đồng độ phức tạp của việc chèn, mà trước đây là không đổi, với độ phức tạp của tìm kiếm, chúng ta chỉ có được sự bình đẳng của tìm kiếm thành công và không thành công. Kết quả thu được là vô cùng hướng dẫn và cho thấy rằng trước khi tiến hành bất kỳ tối ưu hóa "hiển nhiên" nào, người ta nên cân nhắc rất kỹ những lợi ích và tác hại có thể có mà nó sẽ mang lại.

Sau đây là một ví dụ về triển khai tìm kiếm tuần tự trong danh sách được sắp xếp, cùng với phiên bản tương ứng của các hàm chèn và khởi tạo:

Chương trình 4.2. Tìm kiếm liên tiếp (402seq-list.c)

```
struct CElem {
    int key;
    DataType data;
```

```

struct CElem *next;
/* ... */
} *head;

void listInit (void) /* Khởi tạo */
{
    head = (struct CElem *) malloc(sizeof *head);
    head->next = NULL;
}

void listInsert (int key, DataType data) /* Thêm một phần tử mới */
{
    struct CElem *p, *q, *r;
    q = (struct CElem *) malloc(sizeof *head),
    r = (p = head)->next;
    while (r != NULL && r->key < key) {
        p = r;
        r = r->next;
    }
    q->key = key;
    q->data = data;
    q->next = r;
    p->next = q;
}

struct CElem *listSearch(int key) /* Tìm kiếm liên tiếp*/
{
    struct CElem *q;
    for (q = head->next; q != NULL && q->key < key; q = q->next);
    if (NULL == q || key != q->key)
        return NULL;
    else
        return q;
}

void listPrint (void) /*Hiển thị danh sách trên màn hình*/
{
    struct CElem *q;
    char buf[9];
    for (q = head->next; q != NULL; q = q->next) {
        sprintf(buf, "%d | %d", q->key, q->data);
        printf("%8s", buf);
    }
}

```

Bài tập

► 4.2. Xác định số lần so sánh trung bình khi tìm kiếm trong danh sách đã sắp xếp. So sánh với tìm kiếm nhất quán cổ điển.

4.2.2. Tìm kiếm liên tục với sự sắp xếp lại

Trong trường hợp chúng ta có thông tin sơ bộ về xác suất truy cập vào từng mục, chúng ta có thể sắp xếp chúng sao cho mục được tìm kiếm thường xuyên nhất ở đầu danh sách, tiếp theo - ngay sau nó, v.v. Sử dụng chiến lược như vậy chứng tỏ cực kỳ hiệu quả, đặc biệt là với phân bố rất không đồng đều, trong đó một số lượng nhỏ các phần tử được tìm kiếm với xác suất rất cao.

Trong trường hợp chúng ta không có thông tin sơ bộ như vậy, chúng ta có thể tự lấy thông tin đó với sự trợ giúp của các quan sát thông kê đơn giản: chỉ cần gắn một máy đo truy cập cho mỗi phần tử là đủ. Mỗi lần chúng ta tìm kiếm, chúng ta cập nhật bộ đếm tìm kiếm. Rõ ràng là sau khi cập nhật, anh ấy cuối cùng có thể di chuyển lên danh sách. Do đó, nên so sánh với mục trước đó trong danh sách, hai mục này có thể được trao đổi. Do có thể có nhiều lần số truy cập trùng lặp trong danh sách nên trong trường hợp trao đổi cần phải so sánh với phần tử tiếp theo trước đó, v.v ... cho đến khi đạt được vị trí chính xác tương ứng với lần số của phần tử mới. Việc sắp xếp lại diễn ra đúng lúc, tương ứng với n , và trong trường hợp xấu nhất có thể xảy ra trường hợp phần tử cuối cùng của danh sách trở thành phần tử đầu tiên, trao đổi với mọi người trước nó [Wirth-1980].

Tuy nhiên, kế hoạch được đề xuất với việc duy trì các bộ đếm đặc biệt tỏ ra cồng kềnh, yêu cầu bộ nhớ bổ sung về thứ tự của n và có thể được tối ưu hóa. Một lần nữa, nó chỉ ra rằng đơn giản là tốt hơn. Thật vậy, trong trường hợp này, một chiến lược tổ chức lại đơn giản hơn đã tỏ ra cực kỳ hiệu quả. Không có quầy nào được duy trì và không có số liệu thống kê nào được lưu giữ. Thay vào đó, mỗi lần tìm kiếm thành công một mục sẽ đưa nó lên đầu danh sách.

Tất nhiên, chiến lược này không đảm bảo cho chúng ta sự sắp xếp tối ưu của các phần tử theo tần suất truy cập của chúng, nhưng nó dễ bảo trì và đủ hiệu quả. Thật vậy, không giống như tùy chọn

bộ đếm, ở đây việc tổ chức lại diễn ra theo thời gian. Mặt khác, một mục càng được truy cập thường xuyên thì càng có nhiều khả năng nằm trong số các mục đầu tiên trong danh sách. Đồng thời, các đặc thù của địa phương cũng được tính đến tốt hơn: nếu một mục hiếm khi được tìm kiếm trên toàn cầu thường xuyên được tìm kiếm tại một thời điểm nhất định, thì chiến lược này sẽ cho phép chúng ta tận dụng lợi thế của nó. (So sánh với các phương pháp nén thích ứng của ??)

Chương trình 4.3. Tìm kiếm liên tiếp với sự sắp xếp lại (403reorder.c)

```

void listInsert (int key, DataType data) /* Thêm một mặt hàng mới*/
{
    struct CElem *q = (struct CElem *) malloc(sizeof *head);
    q->key = key;
    q->data = data;
    q->next = head;
    head = q;
}

/*Tìm kiếm liên tục với sự sắp xếp lại*/
struct CElem *listSearch(int key)
{
    struct CElem *q, *p = head;
    if (NULL == head)
        return NULL;
    if (head->key == key) return head;
    for (q = head->next; q != NULL; )
        if (q->key != key) {
            p = q;
            q = q->next;
        }
        else {
            p->next = q->next;
            q->next = head;
            return (head = q);
        }
    return NULL;
}

```

Bài tập

- ▷ 4.3. Thực hiện tìm kiếm danh sách sắp xếp lại dựa trên bộ đếm truy cập. So sánh hiệu suất của nó với chuyển đổi không có bộ đếm ở trên.
- ▷ 4.4. So sánh ý tưởng cơ bản của việc tìm kiếm liên tiếp có sắp xếp lại không có bộ đếm với ý tưởng của các phương pháp nén thích ứng (xem ??).
- ▷ 4.5. Hãy xác định số lượng so sánh trung bình trong một tìm kiếm nhất quán với sự sắp xếp lại. So sánh với tìm kiếm nhất quán cổ điển.
- ▷ 4.6. Để so sánh trong trường hợp tốt nhất, trung bình và xấu nhất, ba phương pháp được xem xét để tìm kiếm liên tiếp:
 - a) trong một mảng không có thứ tự (xem 4.2)
 - (b) trong danh sách đã sắp xếp (xem 4.2.1)
 - (c) trong danh sách sắp xếp lại chưa được phân loại (xem 4.2.2)

4.2.

4.3. Tìm kiếm theo từng bước. Tìm kiếm bậc hai

Chúng ta hãy nhìn lại trường hợp của một tập hợp được đặt hàng. Phương pháp tìm kiếm theo danh sách được sắp xếp được thảo luận ở trên hơi khác so với tìm kiếm tuần tự tiêu chuẩn và không sử dụng đủ thứ tự mục. Chúng ta sẽ cố gắng sửa chữa sai lầm này bằng cách thực hiện một cách tiếp cận hoàn toàn khác. Hãy chọn một số bước k và liên tiếp kiểm tra xem khóa của phần tử được tìm kiếm có lớn hơn phần tử đầu tiên hay không, từ phần tử thứ $(k + 1)$, từ phần tử thứ $(2k + 1)$, từ phần tử $(3k + 1)$ phần tử phần tử thứ, ... Tức là chúng ta so sánh nó với $m[1].key$, $m[k + 1].key$, $m[2k + 1].key$, ... Quá trình kết thúc khi đến phần tử lớn hơn hoặc bằng x , hoặc ở cuối mảng.

Chúng ta hãy xem xét kỹ hơn sơ đồ trên (chúng ta sẽ gọi nó là *tìm kiếm theo bước*). Giả sử rằng bằng cách áp dụng nó, chúng ta đã đạt được một phần tử lớn hơn x . Nay giờ chúng ta có thể sử dụng tìm kiếm tuần tự trong khoảng thời gian được xác định bởi hai mốc cuối cùng. Trong trường hợp chúng ta ở ngoài mảng, chúng ta có thể sử dụng tìm kiếm tuần tự từ mốc trước đó đến cuối mảng. Rõ ràng, cách tiếp cận như vậy có thể dẫn đến việc giảm mạnh số lượng mục được xem bởi tìm kiếm nhất quán. Ngoài ra, có thể dễ dàng nhận thấy rằng phương pháp được trình bày là một bản tóm tắt của tìm kiếm tuyến tính. Thật vậy, giá trị thứ hai thu được tại $k = 1$.

Theo mô tả ở trên, phương pháp đề xuất luôn bắt đầu bằng $m[1].key$. Khởi đầu như vậy như thế nào là hợp lý? Không khó để thấy rằng đây thực sự là một lựa chọn tồi, và vì những lý do tương tự mà lựa chọn $m[n].key$ là tệ: nó mang lại cho chúng ta một lượng thông tin tối thiểu. Nó chỉ ra rằng nó là rất thích hợp để bắt đầu trực tiếp với $m[k].key$. (Tại sao?) Dễ dàng thấy rằng trong trường hợp này, tìm kiếm tuyến tính cũng thu được $k = 1$.

Hiệu quả của phương pháp được mô tả tại một k cố định là gì và trường hợp xấu nhất là gì? Rõ ràng là tìm kiếm tuyến tính có cùng giá cho tất cả các khoảng của kiểu $[m[i * k + 1].key; m[(i + 1) * k].key]$, vì nó được thực hiện trên cùng một số phần tử. Ngoại lệ duy nhất có thể là khoảng cuối cùng, có thể chứa ít hơn k phần tử. Trong trường hợp xấu nhất, khóa tìm kiếm nằm trong khoảng cuối cùng, có nghĩa là chúng ta sẽ cần $[n/k]$ so sánh để xác định khoảng mà chúng ta cần áp dụng tìm kiếm tuyến tính. Với những điều này, chúng ta nên thêm độ dài của khoảng, với n , bội số của k , là $k - 1$. Chúng ta nhận được điều đó trong trường hợp xấu nhất khi tìm kiếm với bước k không quá $[n/k] + k - 1$ phép so sánh được thực hiện.

Sau đây là một ví dụ về triển khai thuật toán được mô tả:

Chương trình 4.4. Tìm kiếm theo bước(404jumpsear.c)

```
unsigned seqSearch(unsigned l, unsigned r, int key)
{ while (l <= r)
    if (m[l++].key == key)
```

```

    return l-1;
    return NOT_FOUND;
}

unsigned jmpSearch(int key, unsigned step)
{ unsigned ind;
  for (ind = 0; ind < n && m[ind].key < key; ind += step);
  return seqSearch(ind + 1 < step ? 0 : ind + 1 - step,
                   n < ind ? n : ind, key);
}

```

Câu hỏi quan trọng được đặt ra: Làm thế nào, cho trước n , để chọn k sao cho đảm bảo hiệu quả tối đa? Bảng 4.1 hiển thị số lượng phép so sánh tối đa được thực hiện bằng cách chèn từng bước tại các giá trị khác nhau của n và k :

$n \setminus k$	1	2	3	4	5	6	7	8
1	1							
2	2	2						
3	3	2	3					
4	4	3	3	4				
5	5	3	3	4	5			
6	6	4	4	4	5	6		
7	7	4	4	4	5	6	7	
8	8	5	4	5	5	6	7	8

Bảng 4.1. Số phép so sánh tối đa ở các giá trị khác nhau của n và k .

Có thể thấy rằng các giá trị tốt nhất của k gần với $n/2$, tức là chúng nằm ở giữa hàng tương ứng của bảng hoặc ngay bên trái của nó. Để xác định chính xác hơn giá trị nào của k là tốt nhất và chúng phụ thuộc như thế nào vào n , chúng ta nên xác định sự phụ thuộc nào giữa n và k thì hàm $f(k)$ nhận giá trị nhỏ nhất:

$$f(k) = [n/k] + k - 1.$$

Không khó để chỉ ra (ví dụ, bằng cách nghiên cứu dấu của đạo hàm cấp hai) rằng đây là giá trị của $k = \sqrt{n}$. Khi đó $f(k) \leq 2\sqrt{n} + 1$. Trong trường hợp này phép tìm được gọi là *bậc hai*.

Vì vậy, chúng ta đã thực hiện một cải tiến thực sự tốt: từ n lên \sqrt{n} . Chúng ta có thể đạt được nhiều hơn không? Đúng như dự đoán, câu trả lời cho câu hỏi này là có. Thật vậy, chúng ta đã tối thiểu hóa $f(k)$ ở trên, giả sử rằng sau khi xác định khoảng thời gian, chúng ta sẽ thực hiện tìm kiếm tuần tự. Và đây chính xác là nơi xuất phát ý tưởng: Về điều gì sẽ xảy ra nếu tại thời điểm này chúng ta áp dụng một tìm kiếm khác với một số bước mới l ($1 < l < k$) và chỉ sau đó là một tìm kiếm tuần tự? Thật vậy, độ dài của khoảng sau lần tìm kiếm bước đầu tiên k là \sqrt{n} , có thể là một số đủ lớn để một sự cải thiện như vậy là hợp lý. Bây giờ trong bước đầu tiên, chúng ta sẽ có không quá k phép so sánh để xác định khoảng đầu tiên, sau đó không quá l - để xác định khoảng thứ hai và cuối cùng là không quá $n/(k.l)$ - cho tìm kiếm tuần tự. Chúng ta có thể hình thành một hàm mới để tối thiểu hóa, trong trường hợp đó, dễ dàng hóa ra rằng giá trị nhỏ nhất đạt được đối với $k = l = n/(k.l)$, trong đó $n = k^3$. Trong trường hợp này, thuật toán sẽ thực hiện không quá $3\sqrt[3]{n}$ phép so sánh. Đối với $n \geq 12$, nó chỉ ra rằng $3\sqrt[3]{n} < 2\sqrt{n}$, tức là chúng ta đã đạt được sự cải thiện [Gregory, Rawlins-1997].

Về điều gì sẽ xảy ra nếu chúng ta áp dụng thuật toán lần thứ ba, thứ tư, v.v.? Lập luận tương tự như trên dẫn chúng ta đến giới hạn của độ phức tạp lôgarit. Mặc dù tầm quan trọng của bản tóm tắt được mô tả (chúng ta đã đạt được độ phức tạp lôgarit!) Chúng ta sẽ không tập trung vào nó nữa, vì có một cách đơn giản hơn để đạt được độ phức tạp như vậy.

Bài tập

► 4.7. Hãy thực hiện tìm kiếm hai giai đoạn với các bước k và l . Thực nghiệm tìm các giá trị tối ưu của k và l dưới dạng các hàm của n .

► 4.8. Chứng minh rằng bước tốt nhất trong tìm kiếm bậc hai trong một mảng có n phần tử là n .

4.4. Tìm kiếm nhị phân

Trong trường hợp một tập hợp có số lượng mục lớn và số lượng tìm kiếm lớn, sẽ thuận tiện khi sử dụng tìm kiếm nhị phân sau khi đã phân loại trước các mục. Tìm kiếm nhị phân là một ví dụ điển hình về việc áp dụng nguyên tắc chia và quy tắc La Mã, sẽ được thảo luận chi tiết trong Chương ???. Ý tưởng chính là chia bài toán phức tạp thành nhiều bài toán đơn giản hơn, từ đó có thể được chia thành các bài toán đơn giản hơn. và Quá trình tiếp tục cho đến khi đạt được một bài toán đủ đơn giản với một giải pháp tầm thường.

Tình hình chính xác với tìm kiếm nhị phân là gì? Chúng ta giả sử chúng ta có một mảng được sắp xếp. Ý tưởng là chia mảng thành hai mảng con và xác định xem phần tử được yêu cầu chắc chắn không nằm trong đó. Mảng phụ được đề cập không được xem xét thêm và các nỗ lực tiếp theo sẽ được tập trung vào mảng có triển vọng hơn. Có thể lập luận rằng tìm kiếm nhị phân không phải là một ví dụ cổ điển về việc áp dụng chiến lược chia để trị, vì nó liên quan đến cả hai mảng con. Tuy nhiên, điều trên có thể thấy rằng nó không cấm việc cắt tỉa sớm các tập hợp con không tăng trưởng. Cách phân chia như thế nào? Hãy biểu thị phần tử tìm kiếm là x và giả sử rằng mảng đã được sắp xếp. Hãy so sánh x với phần tử trung bình của nó (Với số phần tử chẵn có hai phần tử trung bình và chúng ta chọn ai không quan trọng). Trong trường hợp hòa, cuộc tìm kiếm kết thúc thành công. Nếu x nhỏ hơn phần tử ở giữa, chúng ta có thể xác định ngay mảng con vô vọng. Thật vậy, đây là những phần tử ở bên phải của giữa: vì mảng được sắp xếp, chúng đều lớn hơn giá trị trung bình và đến lượt nó, anh ta đã lớn hơn x . Một cách tiếp cận tương tự được thực hiện khi x lớn hơn phần tử ở giữa, khi đó mảng con bên trái bị loại bỏ. Quá trình tương tự sau đó được áp dụng cho mảng con không bị từ chối. Ở mỗi bước, chúng ta chia mảng phôi cảnh thành hai mảng con với số phần tử xấp xỉ bằng nhau. Quá trình kết thúc thành công khi tìm thấy phần tử được yêu cầu hoặc đến được mảng trống. Không khó để thấy rằng quá trình này luôn luôn kết thúc. Thật vậy, ở mỗi bước, mảng được đề cập giảm ít nhất một nửa. Ít nhất, vì phần tử giữa của mảng luôn bị từ

chối và điều này với số phần tử lẻ có nghĩa là chúng bị từ chối bởi một phần tử nhiều hơn phần tử còn lại. Nhân tiện, thực tế là ít nhất một phần tử luôn bị từ chối, cụ thể là phần tử ở giữa, là điều cần thiết, bởi vì một trong hai tập hợp (trái hoặc phải) có thể trống. Tuy nhiên, trong trường hợp này cũng vậy, số lượng phần tử trong mảng con được đề cập sẽ giảm mạnh (ít nhất là 1).

Thực hiện trực tiếp các hướng dẫn ở trên, chúng ta nhận được cách triển khai đệ quy rõ ràng sau đây của tìm kiếm nhị phân (được gọi với binSearch (`key who_search, 0, n-1`)):

Chương trình 4.5. Tìm kiếm nhị phân(405binsear0.c)

```
unsigned binSearch(int key, int l, int r)
{ int mid;
  if (l > r)
    return NOT_FOUND;
  mid = (l + r) / 2;
  if (key < m[mid].key)
    return binSearch(key,l,mid-1);
  else if (key > m[mid].key)
    return binSearch(key,mid+1,r);
  else { return mid; }
}
```

Quá trình được mô tả ở trên thường là đệ quy. Tuy nhiên, trong thực tế, chính xác một trong hai tập con được xem xét ở mỗi bước. Do đó, không khó để triển khai một giải pháp lặp lại tương ứng:

Chương trình 4.6. Tìm kiếm nhị phân(406binsear1.c)

```
unsigned binSearch(int key)
{ int l = 0, r = n-1, mid;
  while (l <= r) {
    mid = (l + r) / 2;
    if (key < m[mid].key)
      r = mid - 1;
    else if (key > m[mid].key)
      l = mid + 1;
    else
      return mid;
  }
}
```

```

    return NOT_FOUND;
}

```

Hàm trên sử dụng hai chỉ số l và r , lần lượt chỉ ra ranh giới bên trái và bên phải của khu vực được xem xét. Sự phân chia của từng bước diễn ra ở giữa khu vực giữa. Hàm trả về vị trí của phần tử có khóa value hoặc -1 trong trường hợp không thành công. Vì tại mỗi bước của thuật toán, mảng con được xem xét giảm ít nhất một nửa, thuật toán trên thực hiện không quá $\lceil \log_2 n \rceil + 1$ phép so sánh. Đây là giới hạn trên cho cả tìm kiếm thành công và không thành công [Wirth-1980].

Hãy cố gắng cải thiện phiên bản trên của chương trình. Rõ ràng, ý tưởng cơ bản của tìm kiếm nhị phân "đóng đinh" giới hạn trên của phép so sánh $\lceil \log_2 n \rceil + 1$. Tuy nhiên, vẫn có chỗ để cải thiện. Chúng ta sẽ bắt đầu với một sửa đổi đơn giản về việc thực hiện chương trình được đề xuất và đặc biệt là cách thiết lập ranh giới của khu vực được xem xét. Ở trên, điều này được thực hiện với sự trợ giúp của cặp chỉ số l và r , lần lượt cho biết các đầu bên trái và bên phải của nó. Nay giờ chúng ta sẽ chuyển sang một biểu diễn mới, sử dụng $offset = r - l$ thay vì r . Chúng ta sẽ muốn một sự bù đắp khác luôn là mức độ của hai vợ chồng. Trong trường hợp kích thước của khu vực ban đầu là một mức độ của cặp, thuộc tính này rõ ràng sẽ được duy trì bởi chính nó mà không cần nỗ lực bổ sung từ phía chúng ta. Nếu không, chúng ta sẽ phải bảo mật nó. Cách đơn giản nhất để xử lý vấn đề là trong bước đầu tiên chia mảng thành hai vùng, vùng này phải có kích thước, độ lớn của cặp. Điều này không khó thực hiện, mặc dù nó sẽ vượt qua các khu vực ngay từ bước đầu tiên. Ví dụ: nếu kích thước của mảng là 1000, giải pháp tốt là trước tiên hãy chia nó thành các vùng sau: $(1, 2, \dots, 512)$ và $(489, 490, \dots, 1000)$.

Cả hai vùng đều có cùng kích thước, cụ thể là 512, là lũy thừa lớn nhất của 2, không vượt quá kích thước của mảng $n = 1000$. Các phần tử $489, 490, \dots, 512$ thuộc cả hai vùng. Tuy nhiên, ở mỗi bước tiếp theo, kích thước sẽ vẫn là một mức độ của cặp, với hiệu quả tổng thể của thuật toán dự kiến sẽ được cải thiện. Ở đây, phép toán phân chia nặng được thay thế bằng một chút dịch chuyển sang bên

phải, dẫn đến cải thiện hơn nữa. (Trên thực tế, điều đó gây ra một chút tranh cãi với các bộ vi xử lý ngày nay.)

Chương trình 4.7. Tìm kiếm nhị phân(407binsear2.c)

```

unsigned getMaxPower2(unsigned k)
{ unsigned pow2;
  for (pow2 = 1; pow2 <= k; pow2 <<= 1);
  return pow2 >> 1;
}

unsigned binSearch(int key)
{ unsigned i, l, ind;
  i = getMaxPower2(n);
  l = m[i].key >= key ? 0 : n - i + 1;
  while (i > 0) {
    i = i >> 1;
    ind = l + i;
    if (m[ind].key == key)
      return ind;
    else if (m[ind].key < key)
      l = ind;
  }
  return NOT_FOUND;
}

```

Chúng ta có thể loại bỏ một so sánh trong chu kỳ. Mặt khác, chúng ta sẽ không còn có thể kết thúc sớm thuật toán khi phần tử được tìm kiếm được tìm thấy trước bước cuối cùng. Do đó, sự thay đổi hóa ra có phần gây tranh cãi:

Chương trình 4.8. Tìm kiếm nhị phân(408binsear3.c)

```

unsigned binSearch(int key)
{ unsigned i, l;
  i = getMaxPower2(n);
  l = m[i].key >= key ? 0 : n - i + 1;
  while (i > 1) {
    i = i >> 1;
    if (m[l+i].key < key)
      l += i;
  }
}

```

```
return (l < MAX && m[++l].key == key ? l : NOT_FOUND);  
}
```

Chúng ta hãy xem xét kỹ hơn ý tưởng của thuật toán. Không khó để nhận thấy rằng chúng ta thực hiện tùy chọn nào trong số các tùy chọn trên, đối với mỗi giá trị của kích thước *n* của miền, thứ tự của các phép so sánh liên tiếp cho mỗi bước được xác định trước. Sau đó, chúng ta có thể viết chúng một cách rõ ràng, loại bỏ nhu cầu phân chia và một chu trình nói chung. Chương trình kết quả hóa ra cực kỳ hiệu quả. Bentley đã báo cáo sự cải tiến khoảng 4,5 lần so với phiên bản cổ điển ban đầu (xem [Bentley-1986]). Tất nhiên, cần lưu ý hạn chế rõ ràng là để vẽ vòng tuần hoàn trong các công trình có điều kiện liên tiếp, kích thước của khu vực ít nhất phải được biết gần đúng. Ngoài ra, sự xuất hiện của chương trình có thể giết chết bất kỳ sự nhiệt tình nào:

Chương trình 4.9. Tìm kiếm nhị phân(409binsear4.c)

```
unsigned binSearch(int key)  
{ unsigned l = 0;  
    if (m[512].key < key) l = 1000-512+1;  
    if (m[l+256].key < key) l += 256;  
    if (m[l+128].key < key) l += 128;  
    if (m[l+ 64].key < key) l += 64;  
    if (m[l+ 32].key < key) l += 32;  
    if (m[l+ 16].key < key) l += 16;  
    if (m[l+ 8].key < key) l += 8;  
    if (m[l+ 4].key < key) l += 4;  
    if (m[l+ 2].key < key) l += 2;  
    if (m[l+ 1].key < key) l += 1;  
    return (l < 1000 && m[++l].key == key ? l : NOT_FOUND);  
}
```

Mặc dù tìm kiếm nhị phân cực kỳ hiệu quả, nó nên được sử dụng một cách thận trọng. Nhược điểm chính của nó là nó nhất thiết phải yêu cầu quyền truy cập trực tiếp vào các phần tử của tập hợp, điều này hạn chế chúng ta sử dụng một mảng và không cho phép bất kỳ cấu trúc động nào. Đến lượt mình, yêu cầu điều chỉnh các phần tử của tập hợp lại tạo ra các vấn đề lớn hơn cho việc chèn. Thật vậy,

nếu chúng ta chèn các phần tử mới vào cuối mảng, chúng ta sẽ phải sắp xếp nó trước mỗi lần tìm kiếm, lần chèn tiếp theo, điều này rõ ràng là vô cùng bất hợp lý. Cần có chèn để giữ cho mảng được sắp xếp. Thao tác này rất kém hiệu quả, đặc biệt là với giá trị nhỏ hơn của khóa của phần tử được chèn ($n + 1$), vì điều này dẫn đến việc dịch chuyển tất cả các phần tử lớn hơn theo một vị trí sang bên phải. Điều này đòi hỏi sự dịch chuyển trung bình là $n/2$ và trong trường hợp xấu nhất là n phần tử (n là số phần tử trước khi chèn).

Ở trên, chúng ta đã xem xét rõ ràng rằng tập hợp không cho phép lặp lại khóa. Tuy nhiên, không khó để thấy rằng ngay cả khi chúng ta cho phép lặp lại, tìm kiếm nhị phân sẽ hoạt động trở lại. Thực vậy, nó sẽ lại tìm thấy chính xác một phần tử, nếu một phần tử đó tồn tại. Nay giờ nó vẫn còn để xem xét rằng mảng đã được sắp xếp. Sau đó, các phần tử khác có khóa này phải được đặt theo một hoặc cả hai hướng, bên cạnh phần tử tìm được. Ý tưởng tương tự có thể được sử dụng để giải quyết vấn đề tổng quát hơn là tìm tất cả các phần tử nằm trong một khoảng cho trước.

Các vấn đề về nhu cầu di chuyển một số lượng lớn các mục cũng xảy ra khi xóa. Trên thực tế, khi xóa một số lượng nhỏ các mục, bạn chỉ cần đánh dấu chúng là đã xóa mà không cần di chuyển. Khi tìm kiếm, chúng nên được bỏ qua một cách thích hợp. Tốt nhất là vẫn sử dụng các khóa của họ trong quá trình tìm kiếm và chỉ trong trường hợp hoàn thành thành công, để kiểm tra xem mục tìm thấy có bị xóa hay không. Sau đó, thực hiện tìm kiếm khả thi giữa các phần tử lân cận của nó theo cả hai hướng.

Bài tập

► 4.9. Tại các giá trị nào của n , tìm kiếm nhị phân sẽ nhanh hơn 10, 100, 1000 lần so với tìm kiếm tuần tự cổ điển (xem 4.2)?

► 4.10. Để so sánh về mặt lý thuyết và thực nghiệm các biến thể khác nhau của tìm kiếm nhị phân.

4.5. Tìm kiếm Fibonacci

Mặc dù thuật nhìn có vẻ lạ, nhưng đôi khi chúng ta có thể đạt được tốc độ tìm kiếm cao hơn nếu từ bỏ việc chia thành hai phần *bằng nhau*. Dưới đây chúng ta sẽ xem xét hai sự thay đổi của nhu cầu nhị phân, mỗi sự thay đổi dựa trên một sự cân nhắc cụ thể. Thật vậy, nhược điểm chính của tìm kiếm nhị phân là phép toán chia nặng được thực hiện ở mỗi bước của thuật toán. Thật không may, tách làm đôi là một thao tác khó và có thể làm chậm thuật toán. Do đó, chúng ta khuyến khích khi phát triển các ứng dụng thực tế đặt hàng

$$\text{mid} = (\text{l} + \text{r}) / 2;$$

được thay thế bởi

$$\text{mid} = (\text{l} + \text{r}) >> 1;$$

Sửa đổi được đề xuất sử dụng đáng kể sự biểu diễn bên trong của các số trong hệ thống số nhị phân và thực tế là phép chia 2 có thể được thay thế bằng cách loại bỏ chữ số cuối cùng của số bị chia, tức là bằng cách dịch chuyển một vị trí sang bên phải. Hầu như tất cả các bộ vi xử lý hiện đại đều có hướng dẫn máy như vậy. Chúng ta sẽ nhắc nhớ rằng đối với bất kỳ hệ thống số nào với cơ số p là hợp lệ: Thương trong phép chia số nguyên của p với tư và phần dư có thể được tìm thấy bằng cách loại bỏ chữ số cuối cùng của phép chia (phần dư của phép chia).

Mặc dù rất khó xảy ra, nhưng có thể chiếc máy bạn đang sử dụng không có hướng dẫn thích hợp để chuyển sang phải một chút (các tác giả không biết về chiếc máy như vậy). Trong trường hợp này, sử dụng cái gọi là tìm kiếm Fibonacci là thích hợp. Nhớ lại rằng số Fibonacci được cho bởi công thức truy hồi (xem ??):

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}, n \geq 2.$$

Tìm kiếm Fibonacci rất giống với nhị phân - mảng được chia thành hai phần, một trong số đó bị loại trừ khỏi việc xem xét thêm. Sự khác biệt duy nhất là cách chúng ta chọn mặt hàng để so sánh. Giả sử rằng việc tìm kiếm diễn ra trong một mảng n phần tử, chẳng hạn như $n = F_{k-1}$. Trong bước đầu tiên, chúng ta so sánh khóa yêu

cầu x với $m[F_{k-1}].key$. Trong trường hợp hòa, cuộc tìm kiếm kết thúc thành công. Trong trường hợp $x < m[F_{k-1}]$. Khóa ở bước tiếp theo ta xét các phần tử $1, 2, \dots, F_{k-1} - 1$. Nếu $x > m[F_{k-1}]$. Khóa ở bước tiếp theo, các phần tử $F_{k-1} + 1, F_{k-1} + 2, \dots, n = F_k$ vẫn được xem xét. Người ta tính trực tiếp rằng trong trường hợp đầu tiên cho bước tiếp theo, chúng ta thu được một dãy có độ dài $F_{k-1} - 1$, và trong trường hợp thứ hai - $F_{k-2} - 1$. Quá trình được lặp lại trên phần không bị từ chối cho đến khi tìm thấy phần tử cần thiết hoặc đạt được dãy trống (xem [Horowitz-1977]).

Khả năng thực hiện tìm kiếm Fibonacci trông như thế này ($l \geq 0$ và sao cho $F_{k+l} = n + 1$ và $F_{k+1} > n + 1$):

Chương trình 4.10. Tìm kiếm fibonacci(410fibsear.c)

```
unsigned fib[MAX]; /* Số Fibonacci không vượt quá n*/
```

```
unsigned findFib(unsigned n)
```

```
{ unsigned k;
    fib[0] = 0;
    fib[1] = 1;
    for (k = 2; ; k++)
        if ((fib[k] = fib[k-1] + fib[k-2]) > n)
            return k-1;
    return 0;
}
```

```
unsigned fibSearch(int key)
```

```
{ int p,q,r,k;
    k = findFib(n);
    p = fib[k-1];
    q = fib[k-2];
    r = fib[k-3];
    if (key > m[p].key)
        p += n - fib[k] + 1;
    while (p > 0)
        if (key == m[p].key)
            return p;
        else
            if (key < m[p].key)
                if (0 == r)
                    p = 0;
```

```
    else {
        int t;
        p -= r;
        t = q;
        q = r;
        r = t-r;
    }
else
    if (1 == q)
        p = 0;
    else {
        p += r;
        q -= r;
        r -= q;
    }
return NOT_FOUND;
}
```

Hiệu quả của tìm kiếm Fibonacci là gì? Hóa ra lợi ích của việc xóa bỏ sự chia rẽ không lớn. Giống như tìm kiếm nhị phân, thứ tự thực hiện so sánh được xác định trước cho mỗi giá trị của n. Trong thực tế, nhu cầu Fibonacci xây dựng một cây. Dễ dàng nhận thấy sự chênh lệch chiều cao của các cây con đối với mỗi ngọn nhiều nhất là 1, tức là cây đã cân đối. Loại cây này được gọi trong tài liệu là cây Fibonacci và đã được thảo luận trong phần ??, nơi chúng ta đã chỉ ra rằng chúng là trường hợp xấu nhất của cây cân bằng, cao hơn 45% so với cây cân bằng lý tưởng tương ứng. Do đó, trong trường hợp xấu nhất, nhu cầu Fibonacci khám phá gỗ, cao hơn 45% so với nhu cầu nhị phân (xem 4.4), Và do đó chứng tỏ kém hiệu quả hơn.

Bài tập

- ▷ 4.11. Để so sánh về mặt lý thuyết và thực nghiệm tìm kiếm nhị phân và Fibonacci.

4.5.

4.6. Tìm kiếm nội suy

Khi một người tìm kiếm "Bonev" trong danh bạ điện thoại, anh ta sẽ nhìn vào đầu danh bạ và khi tìm kiếm "Bonev" - ở cuối. Chúng ta không thể tối ưu hóa nhu cầu nhị phân theo cách tương tự (xem 4.4)?

Chúng ta hãy xem xét kỹ hơn. Trong tìm kiếm nhị phân, chúng ta chọn phần tử giữa là mid theo công thức $mid = (l + r)/2$. Hãy viết nó ra theo một cách khác:

$$mid = 1 + (r - 1)/2 \quad (*)$$

Công thức kết quả cho thấy rằng vị trí phân chia tiếp theo của khoảng nhận được bằng cách thêm vào nửa đầu của độ dài của nó. Trong trường hợp danh bạ điện thoại, một danh bạ không hoạt động trên công thức này. Thay vì ở giữa, anh ta lần lượt tìm kiếm ở đầu hoặc cuối thư mục. Điều này có thể được biểu thị bằng công thức kiểu (*), cho mục đích $1/2$ được thay thế bằng một số thích hợp khác. Làm thế nào để chọn số này? Hãy quay lại danh bạ điện thoại. Có một người cố gắng xác định vị trí gần đúng của tên được tìm kiếm, bắt đầu từ chữ cái đầu tiên của nó, biết các chữ cái đầu tiên và cuối cùng của bảng chữ cái. Theo cách tương tự, chúng ta có thể cố gắng tìm vị trí gần đúng của bất kỳ phần tử x nào, biết giá trị của x , cũng như các giá trị của đầu và cuối của khoảng được xem xét. Rõ ràng là vị trí tương đối này có thể được đưa ra bằng công thức nội suy:

$$mid = 1 + k * (r-1),$$

ở đâu

$$k = (x - m[1].key) / (m[r].key - m[1].key)$$

Hệ số k , thay thế cho hằng số $1/2$, thực sự cho chúng ta vị trí gần đúng của phần tử cần thiết trong khoảng được xem xét. Trong việc thực hiện nội suy tìm kiếm cần tính đến một số tính năng cụ thể liên quan đến việc tính giá trị của k . Trước hết, chúng ta nên tránh chia cho 0, trong trường hợp giá trị của tất cả các khóa từ khoảng $[l, r]$ được xem xét trùng nhau. Thứ hai, chúng ta nên đảm bảo rằng $k \in [0, 1]$. Thật vậy, nếu không thì giá trị của giữa sẽ vượt ra ngoài

giới hạn của khoảng $[l, r]$ được xem xét. Khi k vượt quá $[0, 1]$, chúng ta có thể giả sử một cách an toàn rằng phần tử bắt buộc bị thiếu.

Hiệu quả của cải tiến được coi là gì? Trong trường hợp phân phôi đều, tìm kiếm nội suy cho kết quả thực sự đáng chú ý! Khi đó giá trị gần đúng trên của k hóa ra lại rất gần với vị trí thực của phần tử được tìm kiếm và nó được tìm thấy nhanh hơn nhiều so với tìm kiếm nhị phân. Nó được chứng minh rằng khi đó tìm kiếm nội suy thực hiện ít hơn $\log_2(\log_2 n) + 1$ so sánh trong cả tìm kiếm thành công và không thành công. Để người đọc có thể đánh giá hiệu quả của phương pháp đã trình bày, chúng ta sẽ chỉ ra rằng hàm $\log_2(\log_2 n)$ phát triển cực kỳ chậm. Ví dụ $\log_2(\log_2 1000000000) < 5$.

Chương trình 4.11. Tìm kiếm nội suy(411interpol.c)

```
unsigned interpolSearch(int key)
{ unsigned l, r, mid;
  float k;
  l = 0; r = n - 1;
  while (l <= r) {
    if (m[r].key == m[l].key)
      if (m[l].key == key)
        return l;
      else
        return NOT_FOUND;
    k = (float) (key - m[l].key) / (m[r].key - m[l].key);
    if (k < 0 || k > 1)
      return NOT_FOUND;
    mid = (unsigned)(l + k*(r-l) + 0.5);
    if (key < m[mid].key)
      r = mid - 1;
    else if (key > m[mid].key)
      l = mid + 1;
    else
      return mid;
  }
  return NOT_FOUND;
}
```

Mặc dù có những ưu điểm không thể phủ nhận của tìm kiếm nội suy, nó nên được sử dụng hết sức thận trọng. Trước hết, không

nên quên rằng nó là một cải tiến (rất tốt) của nhu cầu nhị phân, đó là lý do tại sao nó kế thừa những thiếu sót chính và khó chịu nhất liên quan đến yêu cầu truy cập trực tiếp và quy định các phần tử của tập hợp. Thứ hai, cần lưu ý những yêu cầu bổ sung và những thiếu sót cần thiết (cái gì cũng có giá cả!). Một mặt, các khóa bắt buộc phải là số hoặc để dễ hiểu như vậy, sao cho k luôn nằm trong phạm vi $[0, 1]$. Nếu không, phương pháp này có thể không áp dụng được. Thứ ba, cần đặc biệt chú ý đến thực tế là các số đầu phẩy động được sử dụng ở đây và phép toán chia số học nặng được sử dụng. Thật không may, chúng ta không thể loại bỏ phép chia ở đây dễ dàng như trong tìm kiếm nhị phân: một mặt, phép chia không còn là 2, và mặt khác, nó không phải là số nguyên.

Yêu cầu rằng các khóa của các phần tử được phân bổ đồng đều cũng trở nên cực kỳ quan trọng. Nếu không đúng như vậy, thì tốc độ của nó có thể giảm xuống đáng kể so với tốc độ của tìm kiếm nhị phân, tiến tới tuyếng tinh. Không khó để xây dựng một ví dụ như vậy. Với mục đích này, chúng ta hãy thử tìm số 2 trong tập hợp sau:

1
10000

Ngoài ra, đối với các giá trị nhỏ của n , các số $\log_2 n$ và $\log_2(\log_2 n)$ không khác nhau đáng kể, do đó, người ta thường coi việc sử dụng tìm kiếm nội suy là không đáng để mạo hiểm. Mặt khác, đối với các tệp đặc biệt lớn, khóa lớn hoặc bộ trí dữ liệu bên ngoài, khi việc so sánh đặc biệt tốn kém, thì nên ưu tiên tìm kiếm nội suy.

Bài tập

► 4.12. Hãy so sánh về mặt lý thuyết và thực nghiệm tìm kiếm nhị phân và nội suy.

4.7. Câu hỏi và bài tập

► 4.13. Tìm tất cả các mục bằng một chia khóa.

Hãy sửa đổi các thuật toán tìm kiếm được đề xuất để chúng không chỉ tìm thấy một, mà là tất cả các phần tử có khóa.

► 4.14. *Tìm kiếm trong ma trận.*

Thực hiện một hàm tìm một số trong ma trận kích thước $n \times m$.

► 4.15. *Tìm kiếm trong một chuỗi ký tự.*

Hãy triển khai một hàm tìm kiếm một chuỗi ký tự trong một mảng chuỗi ký tự nhất định.

► 4.16. *Phạm vi tiếp cận.*

Cho trước một mảng các số tự nhiên, một phần tử có khóa k (k - số tự nhiên) và một số tự nhiên d . Để đề xuất một thuật toán và thực hiện một hàm để tìm tất cả các phần tử x trong mảng mà $|k - x.key| \leq d$.

CHƯƠNG 5

LÝ THUYẾT ĐỒ THỊ

VÀ THUẬT TOÁN

5.1. Các khái niệm cơ bản	331
5.2. Trình bày và các thao tác đơn giản với đồ thị.....	338
5.2.1. Danh sách các cạnh	338
5.2.2. Ma trận lân cận, ma trận trọng số	339
5.2.3. Danh sách những cạnh kề	340
5.2.4. Ma trận tỷ lệ giữa các đỉnh và cạnh	341
5.2.5. Các thành phần kết nối	341
5.2.6. Xây dựng và các hoạt động đơn giản với đồ thị	342
5.3. Trình duyệt trên đồ thị.....	344
5.3.1. Duyệt theo chiều rộng	345
5.3.2. Duyệt theo chiều sâu	350
5.4. Đường dẫn, chu trình và dòng chảy tối ưu trong đồ thị	353
5.4.1. Các ứng dụng trực tiếp của thuật toán duyệt ..	354
5.4.2. Đường tối ưu trong đồ thị	363
5.4.3. Chu trình	386
5.4.4. Các chu trình Hamilton. Bài toán đường thương mại	
391	
5.4.5. Chu trình Euler	395
5.4.6. Luồng trong đồ thị	400

"Có hai kết quả có thể xảy ra: Nếu kết quả xác nhận giả thuyết, nghĩa là bạn đã thực hiện một phép đo. Nếu kết quả trái với giả thuyết, thì bạn đã thực hiện một khám phá."

E. Fermi

5.1. Các khái niệm cơ bản

Lý thuyết đồ thị là một nhánh của toán học hiện đại đã có sự phát triển ấn tượng trong vài thập kỷ qua. Đồ thị là một trong những cấu trúc dữ liệu trừu tượng hữu ích nhất trong khoa học máy tính. Nhiều vấn đề từ các lĩnh vực khoa học và thực tiễn khác nhau có thể được mô hình hóa bằng đồ thị và được giải quyết bằng cách sử dụng một thuật toán thích hợp trên đó. Vì lý do này, Chúng ta sẽ dành cả một chương cho họ, và đến cuối cuốn sách này, họ sẽ có mặt trong nhiều bài toán khác. Trước khi đưa ra một số ví dụ, Chúng ta sẽ xác định rõ khái niệm.

Định nghĩa 5.1. Một đồ thị có hướng hữu hạn được gọi là một cặp có thứ tự (V, E) , trong đó:

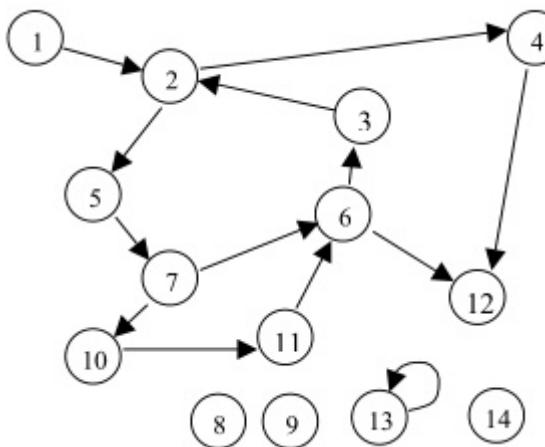
- $V = \{v_1, v_2, \dots, v_n\}$ là một tập hữu hạn các đỉnh;
- $E = \{e_1, e_2, \dots, e_m\}$ là tập hữu hạn các cạnh có định hướng. Mỗi phần tử $e_k \in E (k = 1, 2, \dots, m)$ là một cặp có thứ tự $(v_i, v_j), v_i, v_j \in V, 1 \leq i, j \leq n$.

Ngoài ra, nếu cho một hàm số $f : E \rightarrow R$, phù hợp trên mỗi cạnh e_k trọng số $f(e_k)$ thì đồ thị được gọi là có trọng số. Đôi khi, khi không có nguy cơ mơ hồ, chúng ta sẽ nói chỉ cạnh thay vì cạnh định hướng. Chúng ta sẽ lưu ý rằng một số tác giả sử dụng thuật ngữ cung cho một cạnh có định hướng và cạnh - chỉ cho một cạnh không định hướng.

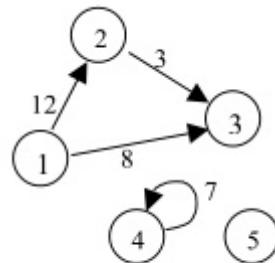
Thông thường, một đồ thị có định hướng được biểu diễn bằng đồ thị trong mặt phẳng bởi một tập hợp các điểm (đường tròn) biểu thị các đỉnh của nó và nối chúng bằng các mũi tên - các cạnh của đồ thị.

Trong Hình 5.1 một đồ thị có 14 đỉnh và 13 cạnh được hiển thị. Chúng ta đã biểu diễn mỗi đỉnh $v_i \in V$ là một đường tròn và chúng ta đặt số của nó - một số tự nhiên duy nhất.

Trong trường hợp điều này sẽ không dẫn đến hiểu lầm bổ sung, thay vì một tập V tùy ý, chúng ta sẽ giả sử rằng V là tập hợp n số tự nhiên đầu tiên. Điều này không làm giảm tính phổ biến của các thuật toán được xem xét và đồng thời dẫn đến một số thuận tiện trong việc triển khai - ví dụ, chúng ta có thể sử dụng các đỉnh làm



Hình 5.1. Đồ thị định hướng.



Hình 5.2. Đồ thị cân định hướng.

chỉ số mảng và các thuật toán khác.

Chúng ta đã biểu diễn mỗi cạnh $(i, j) \in E$ như một mũi tên chỉ từ đỉnh i đến đỉnh j . Lưu ý rằng có thể cho phép một cạnh nhô ra và đi vào cùng một đỉnh: ví dụ, đỉnh 13. Các đường cạnh như vậy được gọi là vòng lặp. Nếu đồ thị có trọng số, trọng lượng của mỗi cạnh sẽ được ghi bên cạnh mũi tên tương ứng, như trong Hình 5.2.

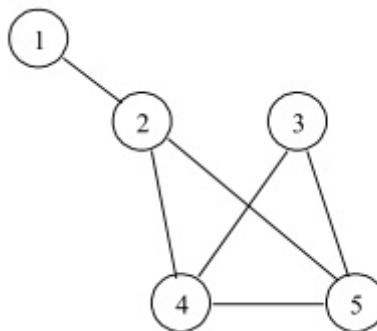
Trong một số trường hợp hiếm hoi, khi xem xét một số bài toán cụ thể, có thể sử dụng các cách lập chỉ mục đỉnh khác (ví dụ: bằng chữ cái Latinh viết thường).

Định nghĩa 5.2. Một đồ thị vô hướng hữu hạn được gọi là một cặp có thứ tự (V, E) , trong đó:

- $V = \{v_1, v_2, \dots, v_n\}$ là một tập hữu hạn các đỉnh
- $E = \{e_1, e_2, \dots, e_m\}$ là tập hữu hạn các cạnh vô hướng. Mỗi phần tử $e_k \in E (k = 1, 2, \dots, m)$ là một cặp không có thứ tự $(v_i, v_j), v_i, v_j \in V, 1 \leq i, j \leq n$.

Ngoài ra, nếu hàm $f(i, j)$ được thiết lập, khớp với giá trị nguyên của mỗi cạnh $(i, j) \in E, f(i, j) = f(j, i)$, đồ thị được gọi là *đồ thị không định hướng có trọng số*.

Trong Hình 5.3 một đồ thị không định hướng được hiển thị - các



Hình 5.3. Đồ thị không định hướng

đỉnh được biểu diễn bằng các đường tròn và mỗi cạnh (i, j) bằng một đoạn thẳng.

Đôi khi chúng ta sẽ muốn coi đồ thị không định hướng $G(V, E)$ là một $G'(V, E')$ có hướng. Trên mỗi đồ thị không định hướng có thể so sánh một đồ thị có hướng tương ứng: trên mỗi cạnh $(i, j) \in E$ các cạnh $(i, j) \in E'$ và $(j, i) \in E'$ trong G' được so sánh. Điều này rất hữu ích khi chúng ta muốn áp dụng một thuật toán hợp lệ cho các đồ thị có định hướng và một đồ thị không định hướng được đưa ra trong câu lệnh bài toán.

Hầu như bất kỳ tập hợp đối tượng nào có kết nối xác định giữa chúng đều có thể được biểu diễn dưới dạng đồ thị. Dưới đây là một số ví dụ:

1) Xem xét bản đồ giao thông của Hà Nội và các ngã ba, tư năm. Chúng có thể được biểu diễn dưới dạng các đỉnh của đồ thị và các đường dẫn trực tiếp giữa chúng - dưới dạng các cạnh. Trọng lượng của các cạnh sẽ là độ dài của các đoạn đường. Ví dụ minh họa là Hình 5.4, trong đó đồ thị là vô hướng, và có thể coi là đồ thị có định hướng.

2) Một mạng máy tính có thể được biểu diễn bằng một đồ thị không định hướng trong đó các máy tính là các đỉnh và mỗi cạnh giữa hai đỉnh cho biết rằng các máy tính tương ứng được kết nối trực tiếp với một mạng.

3) Tập hợp các trang web và các liên kết giữa chúng có thể được biểu diễn dưới dạng một đồ thị có định hướng.



Hình 5.4. Bản đồ giao thông thành phố Hà Nội

4) Một số hợp chất hóa học có thể được biểu diễn dưới dạng các đỉnh của đồ thị không định hướng: Mỗi cạnh sẽ cho biết liệu các hợp chất hóa học tương ứng có thể tương tác hay không. Một ví dụ tương tự là phần ngọn tượng trưng cho các loài cá cảnh và phần cạnh cho biết hai loài cá có thể cùng tồn tại hay không.

5) Các giai đoạn trong quá trình sản xuất một sản phẩm có thể được biểu diễn dưới dạng các đỉnh của một đồ thị có định hướng. Các đường gân chỉ ra công đoạn nào trước ai trong quá trình làm ra sản phẩm.

Nhiều ví dụ khác về đồ thị có thể được đưa ra, cũng như các bài toán được xây dựng trên chúng. Ví dụ: chúng ta có thể quan tâm đến con đường ngắn nhất hoặc rẻ nhất giữa hai ngã tư trong ví dụ 1), hoặc thời gian tối thiểu để tạo ra toàn bộ sản phẩm trong ví dụ 5).

Để xem xét đầy đủ tài liệu hơn nữa, cần phải xác định thêm một

số khái niệm từ lý thuyết đồ thị. Người đọc không cần phải cố gắng ghi nhớ tất cả các định nghĩa ngay bây giờ. Chỉ cần anh ta quay lại đây sau khi kiểm tra phần liên quan và nhớ những gì mình cần là đủ.

Định nghĩa 5.3. Một đồ thị có định hướng (vô hướng) $G(V, E)$ được đưa ra. Nếu cho phép lặp lại trong tập các cạnh của nó (tức là E là một tập đa) thì G được gọi là một *đa đồ thị*.

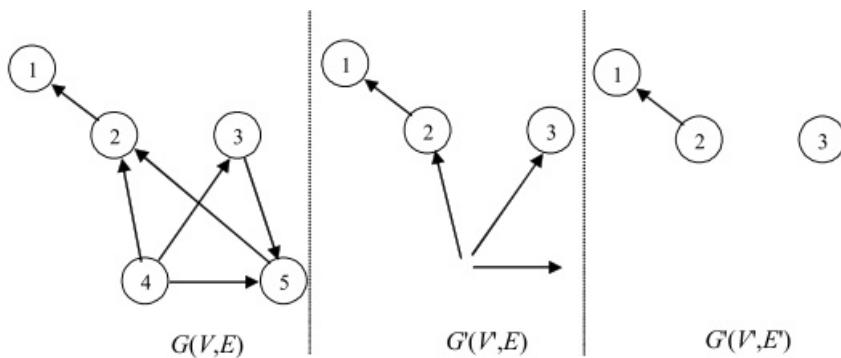
Lưu ý rằng nếu cân đa đồ thị, thì các cạnh khác nhau (i, j) được so sánh với các trọng lượng riêng biệt (nói chung là khác nhau) $f^{(1)}(i, j), f^{(2)}(i, j), \dots$

Định nghĩa 5.4. Một đồ thị có định hướng $G(V, E)$ được cho. Hai đỉnh i và j ($i, j \in V$) được gọi là *kề nhau* nếu ít nhất một trong các cạnh (i, j) và (j, i) thuộc E . Trong trường hợp này, chúng ta cũng nói rằng i và j là điểm đầu các cạnh (i, j) và (j, i) . Đối với mỗi cạnh (i, j) , đỉnh i được gọi là *trước* của j và j là đỉnh *kế tiếp* của i . Mỗi đỉnh i và j được gọi là *chung cạnh* với cạnh (i, j) . Chúng ta nói rằng hai cạnh là *ngẫu nhiên* khi chúng *ngẫu nhiên* có cùng một đỉnh. Trong trường hợp đồ thị vô hướng, các số hạng kè đỉnh, điểm cuối cạnh, tỷ lệ đỉnh và cạnh được định nghĩa tương tự.

Định nghĩa 5.5. Một đồ thị có định hướng $G(V, E)$ được cho. Bán độ ở đầu ra của đỉnh i , $i \in V$ được gọi là số cạnh (i, j) , $j \in V$. Tương tự, số lượng cạnh (j, i) , $j \in V$ được gọi là bán độ ở đầu vào của i . Tổng của nửa giai đoạn đầu vào và nửa giai đoạn đầu ra được gọi là bậc của đỉnh i . Một đỉnh cô lập được gọi, có tung độ bằng 0. Trong một đồ thị không có hướng, tung độ của đỉnh i được gọi là số cạnh (i, j) *ngẫu nhiên* của nó.

Trong Hình 5.1 tung độ của đỉnh 2 là 4 và đỉnh 14 là biệt lập.

Định nghĩa 5.6. Cho là một đồ thị có định hướng (vô hướng) $G(V, E)$ và $V' \subseteq V$. Nếu tất cả các cạnh (i, j) bị loại trừ khỏi E , sao cho $i \in V'$ hoặc $j \in V'$ và phần còn lại được giữ nguyên, thì chúng ta nói rằng $G'(V', E')$ được quy nạp bởi đồ thị con V' (hoặc chỉ đồ thị con) của G (xem Hình 5.5).



Hình 5.5. Được quy nạp bởi tập $V' = \{1, 2, 3\}$ đồ thị con G' của G .

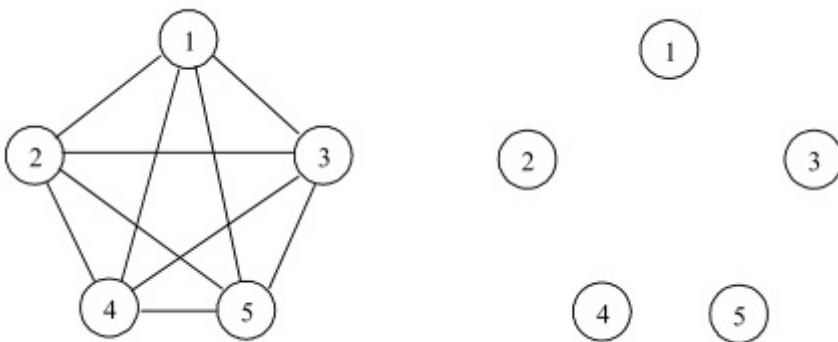
Định nghĩa 5.7. Một đường đi trong đồ thị có định hướng (vô hướng) $G(V, E)$ được gọi là dãy các đỉnh v_1, v_2, \dots, v_k sao cho mỗi $i = 1, 2, \dots, k - 1$ được thỏa mãn (vi, trong $i + 1 \in E$). Các đỉnh v_1 và v_k được gọi là các điểm cuối của con đường. Nếu $v_1 = v_k$, đường đi được gọi là vòng lặp. Nếu với mỗi $i \neq j$ ($1 \leq i, j \leq k$) theo sau $v_i \neq v_j$, thì đường đi được gọi là đơn giản. Theo đó, nếu $v_1 = v_k$ và tất cả các đỉnh khác nhau, thì vòng lặp được gọi là đơn giản. Khi một đồ thị chứa ít nhất một chu trình, nó được gọi là chu trình, nếu không nó được gọi là không có chu trình.

Định nghĩa 5.8. Cho đồ thị $G(V, E)$ và $G'(V, E')$ sao cho với mỗi $i, j \in V, i \neq j$, cặp có thứ tự (i, j) thuộc đúng một trong các tập E' hoặc E . Khi đó G' được gọi là phần bù của G . Một đồ thị không có cạnh được gọi là rỗng (Hình 5.6). Nếu $(i, j) \in E$ với mọi $i, j \in V, i \neq j$ thỏa mãn đối với một đồ thị đã cho thì đồ thị được gọi là hoàn chỉnh.

Lưu ý rằng định nghĩa cuối cùng là đối xứng, tức là nếu G' là phần bù của G thì G là phần bù của G' .

Định nghĩa 5.9. Nếu một đồ thị $G'(V', E')$ với đỉnh t được gây ra bởi G và G' là hoàn chỉnh, chúng ta nói rằng G' là một nhấp chuột của G . T lớn nhất mà G có t -click được gọi là số lần nhấp cho G .

Định nghĩa 5.10. Một đồ thị có định hướng được cho là có tính liên kết yếu nếu cứ hai đỉnh i và j là kết thúc của ít nhất một đường đi (tức là có ít nhất một đường đi từ i đến j hoặc từ j đến i). Khi trong



Hình 5.6. Đồ thị vô hướng hoàn chỉnh và transpose rỗng.

một đồ thị có hướng mà cứ hai đỉnh i, j có một đường đi từ i đến j và từ j đến i , thì đồ thị được gọi là liên thông mạnh. Một đồ thị vô hướng được gọi là liên thông nếu có một đường đi giữa mỗi cặp đỉnh i, j của nó.

Định nghĩa 5.11. Đường kính của đồ thị được gọi là số lớn nhất k , sao cho đường đi nguyên tố ngắn nhất (đường có số đỉnh tối thiểu) giữa hai đỉnh $i, j \in V$ bất kỳ chứa ít nhất k đỉnh.

Ví dụ, nếu chúng ta xem các trang web và liên kết như một đồ thị có định hướng, thì đường kính k của nó là số trang tối đa mà chúng ta phải đi qua, bắt đầu từ một trang ngẫu nhiên và đi theo các siêu liên kết để đến bất kỳ trang nào khác trên web. Các nghiên cứu cho thấy "đường kính của Internet" là khoảng 19 [Web-d].

Định nghĩa 5.12. Thành phần được liên thông mạnh (yếu) trong một đồ thị có định hướng $G(V, E)$ được gọi là bất kỳ đồ thị con $G'(V', E')$ nào mà nó được đáp ứng:

- 1) G' là một đồ thị liên thông mạnh (yếu).
- 2) Không có đồ thị con nào khác được kết nối mạnh (yếu) với $G''(V'', E'')$ của G , sao cho G' là một đồ thị con của G'' .

Tương tự, một thành phần liên thông được xác định trong một đồ thị không định hướng.

Định nghĩa 5.13. Một đồ thị liên thông vô hướng không có vòng lặp được gọi là cây. Nếu chúng ta chọn thêm một ngọn cây cho gốc, thì cấu trúc kết quả được gọi là cây gốc.

Định nghĩa này là một sự thay thế cho định nghĩa được thảo luận trong ?? định nghĩa của cây).

Định nghĩa 5.14. Cây bao phủ trong đồ thị không định hướng liên thông $G(V, E)$ được gọi là bất kỳ đồ thị con chu trình liên thông $G'(V, E')$ nào của G .

Bài tập

▷ 5.1. So sánh các định nghĩa ?? và ??.

5.2. Trình bày và các thao tác đơn giản với đồ thị

Cho đến nay, Chúng ta đã sử dụng hai cách để biểu diễn một đồ thị:

- Trực tiếp theo định nghĩa - qua các tập V và E đã cho.
- Về mặt đồ họa: thông qua tập hợp các điểm (đường tròn) và các kết nối (mũi tên) giữa chúng.

Bản trình bày thứ hai (mặc dù trực quan) không thể áp dụng được và bản trình bày đầu tiên - thường không thuận tiện, đối với quá trình xử lý trên máy tính. Do đó, có nhiều cách khác để trình bày một đồ thị - chúng sẽ là chủ đề của đoạn văn này. Việc lựa chọn cách biểu diễn máy tính thích hợp nhất phụ thuộc vào bài toán cụ thể - tốt nhất là sử dụng một kiểu biểu diễn trong đó các phép toán được thực hiện thường xuyên có độ phức tạp thuật toán thấp hơn (hoặc yêu cầu ít bộ nhớ hơn).

5.2.1. Danh sách các cạnh

Gần nhất với định nghĩa của đồ thị là một biểu diễn với danh sách các cạnh của nó. Chúng ta biểu diễn một đồ thị có định hướng dưới dạng danh sách các cặp có thứ tự (i, j) . Trong trường hợp đồ thị vô hướng, các cặp không có thứ tự. Trong một đồ thị có trọng số

(có định hướng), chúng ta xem xét (có thứ tự) các bộ ba (i, j, k) , trong đó số thực k là trọng số $f(i, j)$ của cạnh tương ứng. Sau đó, chúng ta có thể nhận ra trên C bằng cách sử dụng mảng float $A[3][m]$, với m là số cạnh của đồ thị.

Rõ ràng, trong cách biểu diễn này, bộ nhớ cần thiết sẽ là $\Theta(m)$, đây là mức độ phức tạp của việc kiểm tra xem hai đỉnh có kề nhau hay không, đây được coi là một trong những phép toán phổ biến nhất. Trong trường hợp Chúng ta sắp xếp các cạnh, Chúng ta sẽ có thể áp dụng tìm kiếm nhị phân (xem ??) Và do đó giảm độ phức tạp của kiểm tra xuống) $\Theta(\log_2 m)$. Như chúng ta sẽ thấy bên dưới, có những biểu diễn trong đó độ phức tạp cuối cùng nhỏ hơn nhiều: một hàm của n , và thậm chí là một hằng số. Do đó, biểu diễn này được áp dụng chủ yếu cho các đồ thị thưa, nghĩa là, đồ thị có số cạnh tương đối nhỏ: $m \in O(n)$.

5.2.2. Ma trận lân cận, ma trận trọng số

Ma trận lân cận là một trong những cách được sử dụng phổ biến nhất để biểu diễn một đồ thị trong bộ nhớ. Trong đó, một ma trận vuông $A[n][n]$ được so sánh trên một đồ thị có hướng với n đỉnh. Giá trị của $A[i][j]$ bằng 1 khi tồn tại cạnh (i, j) và $A[i][j] = 0$ - nếu không. Để thuận tiện cho việc thực hiện một số thuật toán, khi cạnh (i, j) không tồn tại mà tồn tại (j, i) thì tại vị trí $A[i][j]$ trong ma trận được viết giá trị -1 (và lần lượt là 1 - của $A[j][i]$).

Nếu đồ thị là vô hướng, thì $A[i][j] = 1$ khi có cạnh (i, j) hoặc (j, i) và $A[i][j] = 0$ nếu không. Với một đồ thị không có trọng số, trong trường $A[i][j]$ không viết 1 mà là số cạnh giữa các đỉnh i và j . Khi đồ thị có trọng số, trọng số $f(i, j)$ được viết ở vị trí $A[i][j]$, và ma trận $A[][]$ được gọi là *ma trận trọng số*.

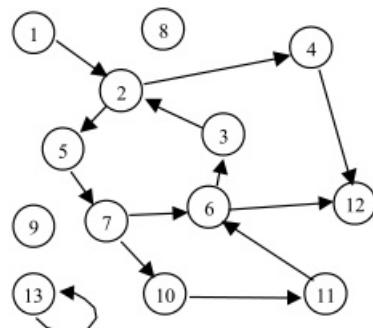
Trong một số bài toán, trọng số cho các đỉnh của đồ thị cũng được xác định. Chúng có thể được viết tại các vị trí $A[i][i]$ cho mỗi đỉnh i của đồ thị. Các trường này từ ma trận sẽ không bị chiếm dụng trừ khi đồ thị không chứa vòng lặp (các cạnh của kiểu (i, i)).

Trong một đồ thị vô hướng, ma trận lân cận là *đối xứng* về đường chéo chính của nó. Trong trường hợp này, có thể áp dụng cách bảo quản cụ thể hơn trong bộ nhớ (ví dụ, tuyển tính hóa, để lưu phần

trùng lặp, chiếm $\frac{n(n - 1)}{2}$ thêm ô nhớ [Rakhnev, Garov, Gavrilov-1995]).

Một ưu điểm quan trọng của biểu diễn ma trận tiệm cận là việc kiểm tra sự tồn tại của một cạnh giữa hai đỉnh được thực hiện bằng một phép toán duy nhất. Mặt khác, việc tìm tất cả những người thừa kế tại một đỉnh có độ phức tạp là $\Theta(n)$, bất kể đỉnh đó có bao nhiêu người thừa kế (tức là ngay cả khi đỉnh có một người thừa kế duy nhất, tất cả n lần kiểm tra vẫn sẽ phải thực hiện). Một ví dụ về đồ thị có định hướng và ma trận lân cận tương ứng của nó được cho trong Hình 5.2.2. Lưu ý rằng trong khi việc đánh số đỉnh bằng đồ thị bắt đầu từ 1, trong việc triển khai C, việc lập chỉ mục của các phần tử bắt đầu từ 0. Tính năng cụ thể này phải được tính đến khi xem xét tất cả các triển khai của các thuật toán trong chương này.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	-1	0	-1	1	1	0	0	0	0	0	0	0	0
2	0	1	0	0	0	-1	0	0	0	0	0	0	0
3	0	-1	0	0	0	0	0	0	0	0	0	1	0
4	0	-1	0	0	0	0	1	0	0	0	0	0	0
5	0	0	1	0	0	0	-1	0	0	0	-1	1	0
6	0	0	0	0	-1	1	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	-1	0	0	0	1	0	0
10	0	0	0	0	0	1	0	0	0	-1	0	0	0
11	0	0	0	-1	0	-1	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	1



Hình 5.7. Ma trận lân cận và đồ thị tương ứng của nó.

5.2.3. Danh sách những cạnh kề

Trong biểu diễn này, một danh sách những người thừa kế của nó được lưu giữ cho mỗi đỉnh i bằng đồ thị. Đối với cột trong Hình 5.2.2. bản trình bày sẽ như thế này:

$1 \rightarrow \{2\}; 2 \rightarrow \{4, 5\}; 3 \rightarrow \{2\}; 4 \rightarrow \{12\}; 5 \rightarrow \{7\}; 6 \rightarrow \{3, 12\}; 7 \rightarrow \{6, 10\}; 8 \rightarrow \{\}; 9 \rightarrow \{\}; 10 \rightarrow \{11\}; 11 \rightarrow \{6\}; 12 \rightarrow \{\}; 13 \rightarrow \{13\}.$

Danh sách liên kết động thường được sử dụng để triển khai máy tính (xem ??). Trong đó, độ phức tạp của việc tìm tất cả những người

thừa kế tại một đỉnh nhất định là tuyển tính về số lượng người thừa kế: một lợi thế so với ma trận lân cận, trong đó việc tìm những người thừa kế có độ phức tạp $\Theta(n)$. Tuy nhiên, nhược điểm là việc kiểm tra xem có một đường cạnh giữa hai đỉnh hay không cũng là một sự phức tạp phụ thuộc tuyển tính vào số lượng người thừa kế.

Cũng có thể thực hiện tinh biếu diễn này: mảng một chiều với kích thước bằng số lượng người thừa kế cho mỗi đỉnh của đồ thị. Trong trường hợp này, việc kiểm tra xem có một cạnh nào giữa hai đỉnh được thực hiện hiệu quả hơn nhiều hay không - với độ phức tạp logarit. Điều này đạt được bằng cách sắp xếp những người thừa kế và sử dụng tìm kiếm nhị phân.

Rõ ràng là cũng như chúng ta có thể duy trì một danh sách những người thừa kế, vì vậy chúng ta có thể xây dựng một danh sách những người tiền nhiệm, trong trường hợp đó, độ phức tạp của các hoạt động cơ bản sẽ giống nhau. Trong thực tế, một bản trình bày với một danh sách những người đi trước được sử dụng rất hiếm khi.

5.2.4. Ma trận tỷ lệ giữa các đỉnh và cạnh

Trong biếu diễn này, một ma trận có kích thước $n \times m$ được sử dụng. Nếu chúng ta xem xét một đồ thị không định hướng $G(V, E)$, với mỗi cạnh $e_k = (i, j)$, cột k sẽ chứa hai đơn vị - tại vị trí i và j , và trong tất cả các phần tử khác trong cột sẽ được viết 0. Đối với một đồ thị có định hướng của vị trí thứ i được viết là 1 và thứ j - thứ -1 , vì tất cả các phần tử khác trong cột là số không. Nếu chúng ta có một vòng lặp (i, i) , thì tại vị trí thứ i của cột e_k được viết 2.

Biểu diễn này có thể áp dụng cho một số bài toán cụ thể (ví dụ: tìm số cây bao phủ trong đồ thị, v.v.), nhưng thường rất hiếm khi được sử dụng vì hai lý do. Trước hết, bộ nhớ cần thiết là $\Theta(m.n)$, trong khi ma trận rất thưa thớt. Kiểm tra xem có một cạnh nào giữa hai đỉnh cũng không hiệu quả và có độ phức tạp $\Theta(m)$.

5.2.5. Các thành phần kết nối

Cách trình bày cuối cùng mà chúng ta sẽ xem xét là thông qua các thành phần kết nối. Nó khác với những cái trước ở chỗ đồ thị,

một khi được biểu diễn trong bộ nhớ, sẽ bị "mất", và không thể khôi phục rõ ràng các tập các đỉnh và các cạnh của nó, tức là chúng ta xem xét sự biểu diễn các thuộc tính nhất định của đồ thị. Một điều kiện bổ sung là đồ thị không được định hướng và không có trọng số.

đồ thị được chia thành các thành phần kết nối, với mỗi tập hợp được ánh xạ thành một tập hợp. Ưu điểm là dễ dàng kiểm tra xem hai đỉnh có được nối với nhau bằng một đường dẫn hay không - điều này được thực hiện nếu và chỉ khi chúng thuộc cùng một tập hợp.

Mảng một chiều $A[n]$ có thể được sử dụng để triển khai máy tính, mảng này chấp nhận các giá trị từ 1 đến n . Và $[i]$ có giá trị k t.s.t.k đỉnh thứ i của đồ thị thuộc thành phần thứ k của kết nối. Biểu diễn này cho phép duy trì một đồ thị không lồ (chỉ bộ nhớ cần thiết $\Theta(n)$), mà nó có thể kiểm tra xem hai đỉnh có được nối với nhau bằng một đường dẫn hay không với một phép so sánh duy nhất: liệu $A[i]$ có bằng $A[j]$, hay không.

Ma trận khả năng tiếp cận (mà chúng ta sẽ thảo luận lại trong 5.4.2) Là một cách khác để biểu diễn các thuộc tính đồ thị thông qua các thành phần kết nối. Tuy nhiên, nó kém hiệu quả hơn nhiều so với những cách được mô tả ở trên. Khi trình bày một đồ thị có ma trận khả năng truy cập, một mảng hai chiều $A[n][n]$ được sử dụng, trong đó giá trị của $A[i][j]$ là 1 nếu có một đường đi giữa các đỉnh i và j , và 0 - nếu không.

5.2.6. Xây dựng và các hoạt động đơn giản với đồ thị

Các thao tác chính liên quan đến việc xây dựng và sửa đổi đồ thị như sau:

- tạo một đồ thị rỗng.
- thêm / bớt đỉnh.
- thêm / bớt một cạnh.

Sau đây là các hoạt động được đề cập trong các điểm trước:

- kiểm tra sự tồn tại của một đỉnh
- kiểm tra sự tồn tại của một cạnh
- tìm những người thừa kế của một đỉnh

Chúng ta sẽ không chỉ rõ cách thực hiện các thao tác này cho từng loại bản trình bày. Ví dụ, Chúng ta sẽ chỉ ra một cách triển khai cho một đồ thị được biểu diễn bằng một ma trận lân cận (ma trận trọng số).

```

/* Số đỉnh tối đa trong đồ thị */
#define MAXN 200

/* Số đỉnh trong đồ thị */
unsigned n;

/* Ma trận trọng số của đồ thị */
int A[MAXN][MAXN];

/* Sự thay đổi trọng lượng của đỉnh i*/
A[i][i] = k;

/* Thêm một cạnh có trọng số k nối các đỉnh i và j*/
A[i][j] = k;
A[j][i] = k; /*nếu đồ thị không có định hướng*/

/* Loại bỏ cạnh nối các đỉnh i và j*/
A[i][j] = 0;

/* Kiểm tra một cạnh giữa các đỉnh i và j*/
if (A[i][j] != 0) { /* có cạnh */ } else { /* không có cạnh */ }

/* Tìm tất cả những cạnh thừa kế ở đầu*/
for (k = 0; k < n; k++) if (k != i) {
    if (A[i][k] != 0) { /* đỉnh k là đỉnh kế tiếp của i */ ; }
}

```

Trước khi kết thúc, chúng ta sẽ so sánh mức độ phức tạp của các thao tác trong các màn trình diễn khác nhau. Trong mỗi bài toán, chúng ta sẽ cố gắng trình bày trong đó các thao tác được thực hiện thường xuyên nhất có độ phức tạp thấp nhất. Có thể trình bày một đồ thị theo nhiều cách tại một thời điểm nếu chúng ta có đủ bộ nhớ và khi điều này dẫn đến việc triển khai thuật toán đã chọn hiệu quả hơn.

Bảng 5.1 có nghĩa là không thể khôi phục duy nhất đồ thị về đồ

Bài thuyết trình Thao tác	Ma trận cạnh kề	Danh sách cạnh thừa kế	Danh sách các cạnh	Ma trận liên thuộc	Thành phần liên thông
Thêm cạnh	$\Theta(1)$	$\Theta(\log_2 n)$	$\Theta(\log_2 m)$	$\Theta(1)$	$\Theta(n \cdot \log_2 n)$
Xóa cạnh	$\Theta(1)$	$\Theta(\log_2 n)$	$\Theta(\log_2 m)$	$\Theta(m)$	\times
Kiểm tra sự tồn tại của cạnh	$\Theta(1)$	$\Theta(\log_2 n)$	$\Theta(\log_2 m)$	$\Theta(m)$	\times
Tìm cạnh thừa kế đỉnh cao	$\Theta(n)$	$\Theta(d_i)$	$\Theta(n + \log_2 m)$	$\Theta(m)$	\times
Kiểm tra kết nối của hai đỉnh với một con đường	$\Theta(n^2)$	$\Theta(n + m)$	$\Theta(n \cdot \log_2 m)$	$\Theta(m^2)$	$\Theta(1)$
Bộ nhớ cần thiết	$\Theta(n^2)$	$\Theta(m)$	$\Theta(m)$	$\Theta(n \cdot m)$	$\Theta(n)$

Bảng 5.1. So sánh mức độ phức tạp của các hoạt động trong các buổi biểu diễn khác nhau.

thì đã chọn trình bày và không thể thực hiện thao tác tương ứng.

Bài tập

- 5.2. Xem xét làm thế nào để đạt được độ phức tạp trong Bảng 5.1 không được đề cập trong đoạn này và theo giả định nào: Ví dụ, để đạt được độ phức tạp $\Theta(\log_2 m)$ để kiểm tra sự tồn tại của một cạnh trong danh sách các cạnh, chúng ta giả sử rằng chúng ta sử dụng tìm kiếm nhị phân trong một danh sách tĩnh các cạnh được sắp xếp theo số đỉnh.

5.3. Trình duyệt trên đồ thị

Bằng cách duyệt qua một đồ thị không định hướng (có định hướng), chúng ta sẽ hiểu được một lần truy cập liên tiếp (xem) mỗi

đỉnh của đồ thị đúng một lần. *Chiến lược trình duyệt xác định thứ tự* mà các đỉnh sẽ được xem xét.

Có hai chiến lược cơ bản để duyệt qua một đồ thị (tùy thuộc vào hai phần tiếp theo), được gọi là duyệt theo chiều rộng và duyệt theo chiều sâu. Tầm quan trọng đặc biệt của hai thuật toán này được xác định bởi:

- Phạm vi rộng của các bài toán mà họ thấy ứng dụng.
- Khả năng giải quyết các vấn đề thuật toán phức tạp bằng cách áp dụng các sửa đổi nhỏ của các thuật toán cơ bản.
- Độ phức tạp thuật toán tốt, đạt được trong hầu hết các trường hợp.

5.3.1. Duyệt theo chiều rộng

Duyệt theo chiều rộng của đỉnh i sẽ được gọi là chiến lược sau để duyệt qua đồ thị: chúng ta bắt đầu từ đỉnh i , xem xét tất cả các lân cận trực tiếp của nó và chỉ sau đó chuyển sang duyệt xa hơn (duyệt theo chiều rộng của từng lân cận của nó). Bằng cách này, đạt được sự đi ngang tuần tự của các "mức", bắt đầu từ đỉnh bắt đầu, trong khi đi qua tất cả các đỉnh của đồ thị, có thể truy cập được bởi i . Duyệt theo chiều rộng của đồ thị được gọi là đi qua tất cả các đỉnh của nó theo cách được mô tả - tức là liên tiếp chọn một đỉnh bắt đầu (ngẫu nhiên) cho đến khi tất cả các đỉnh của đồ thị được duyệt qua.

Từ bây giờ, chúng ta sẽ sử dụng chữ viết tắt $BFS(i)$ (Breadth-First-Search) để biểu thị chức năng duyệt theo chiều rộng của đỉnh i .

Để làm ví dụ, chúng ta sẽ áp dụng chiến lược được mô tả trên đồ thị trong Hình 5.8.

Nếu chúng ta chọn 1 làm đỉnh bắt đầu, kết quả duyệt sẽ như sau:

$BFS(1)$:

cấp độ 1: 1

cấp độ 2: 2

cấp độ 3: 3, 4, 5

cấp độ 4: 7, 6, 12

cấp độ 5: 10

cấp độ 6:11

Nếu thay vì 1, chúng ta chọn 3 cho đỉnh ban đầu, khi thực thi $BFS(3)$, chúng ta nhận được:

$BFS(3)$:

cấp độ 1: 3

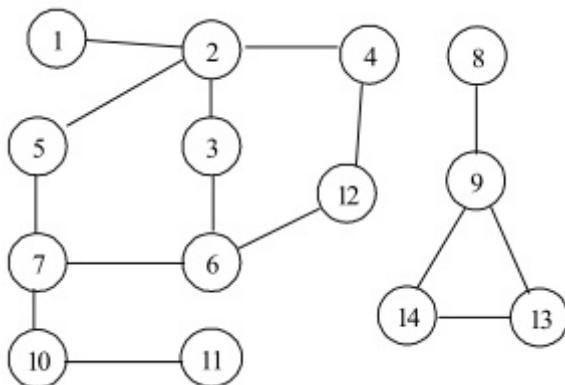
cấp độ 2: 2, 6

cấp độ 3: 1, 4, 5, 7, 12

cấp độ 4:10

cấp độ 5:11

Một số ứng dụng có thể có của việc duyệt theo chiều rộng là đáng chú ý ngay lập tức. Ví dụ, tìm số lượng đỉnh tối thiểu có liên quan giữa hai đỉnh hoặc tìm các thành phần kết nối của đồ thị. Chúng ta sẽ xem xét những điều này và các ví dụ khác xa hơn một chút.



Hình 5.8. Đồ thị không định hướng.

duyệt theo chiều rộng được sử dụng gián tiếp trong các tác vụ khác khi yêu cầu khoảng cách tối thiểu. Ví dụ, phương pháp sóng (xem Bài toán 5.110), áp dụng để tìm đường đi nhỏ nhất giữa hai ô của ma trận, v.v., cũng có thể được coi là đi ngang qua một loại đồ thị đặc biệt theo chiều rộng.

Thực hiện duyệt theo chiều rộng từ một đỉnh nhất định

Chúng ta sẽ biểu diễn đồ thị bằng ma trận lân cận và chúng ta sẽ sử dụng cách biểu diễn này trong hầu hết các bài toán từ đồ thị. Nó

đủ hiệu quả cho hầu hết các tác vụ và đồng thời không làm giảm khả năng đọc của mã với phân bổ bộ nhớ động, triển khai nhiều cấu trúc dữ liệu phức tạp và hơn thế nữa.

Tất nhiên, cả trong quá trình duyệt và các bài toán được thảo luận bên dưới, chúng ta sẽ cố gắng trình bày cho người đọc các giải pháp luôn tốt nhất (theo quan điểm thuật toán) được biết đến cho đến nay.

Trong quá trình duyệt theo chiều rộng, chúng ta phải tìm những người thừa kế của một đỉnh i cho trước. Từ Bảng 5.1, có thể thấy rằng nếu chúng ta sử dụng ma trận lân cận, độ phức tạp sẽ là $\Theta(n)$. Điều này là do đối với mỗi đỉnh n của đồ thị, nó được kiểm tra xem nó có phải là người thừa kế của i hay không (để so sánh, trong trường hợp thực hiện với một danh sách những người thừa kế, chúng ta có thể lấy trực tiếp những người thừa kế của i bằng cách duyệt qua danh sách). Do đó, tổng độ phức tạp của duyệt là $\Theta(m + n)$ - khi trình bày với danh sách lân cận và $\Theta(n^2)$ - khi trình bày với ma trận lân cận.

Chúng ta sẽ duy trì một hàng đợi trong đó ban đầu chỉ có đỉnh xuất phát. Sau đó, trong khi có ít nhất một đỉnh trong hàng đợi, chúng ta thực hiện như sau: chúng ta lấy ra đỉnh ở đầu hàng đợi, nhìn vào nó và thêm vào hàng đợi tất cả những người thừa kế chưa được kiểm tra cho đến nay của nó. Chúng ta sẽ đánh dấu các ngọn là đã truy cập vào thời điểm chúng ta thêm chúng vào hàng đợi:

Thuật toán duyệt theo bề rộng

BFS(i)

```
{
    Chúng ta tạo một hàng đợi trống;
    Chúng ta thêm vào hàng đợi đỉnh  $i$ ;
    for ( $k = 1, 2, \dots, n$ ) used[ $k$ ] = 0;
    used[ $i$ ] = 1;
    while (Hàng đợi không trống) {
         $p$  = Chúng ta xóa phần tử ở đầu hàng đợi;
        Phân tích đỉnh  $p$ ;
        for (với mọi đỉnh kề  $j$  của  $p$ )
            if ( $0 == \text{used}[j]$ ) { /*nếu đỉnh  $j$  không bị bỏ qua*/
                Thêm vào hàng đợi đỉnh  $j$ ;
                used[ $j$ ] = 1; /* chúng ta đánh dấu  $j$  là bỏ qua*/
            }
    }
}
```

```
    }
```

Nhận thức đầy đủ sau đây. Dữ liệu đầu vào được sử dụng trong quá trình triển khai bên dưới được đặt làm hằng số ở đầu chương trình và dành cho cột được hiển thị trong Hình 5.8 Số đỉnh của đồ thị là n và $A[MAXN][MAXN]$ là ma trận lân cận của nó. Đỉnh bắt đầu cho việc duyệt được đặt với hằng số v .

Trong ví dụ minh họa, đồ thị là không có hướng, nhưng chương trình dưới đây sẽ áp dụng chính xác thuật toán trong trường hợp đồ thị có hướng.

Chương trình 5.1. Duyệt theo chiều rộng(501 bfs.c)

```
#include <stdio.h>
/* Số đỉnh tối đa trong đồ thị */
#define MAXN 200
/* Số đỉnh của đồ thị */
const unsigned n = 14;
/* Duyệt theo chiều rộng từ đỉnh v */
const unsigned v = 5;
/* Ma trận kề của đồ thị */
const char A[MAXN][MAXN] = {
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
{0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0},
{0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1}
};

char used[MAXN];
/* Chức năng duyệt chiều rộng từ đỉnh đã cho */
void BFS(unsigned i)
```

```

{ unsigned k, j, p, queue[MAXN], currentVert, levelVertex,
queueEnd;
for (k = 0; k < n; k++) queue[k] = 0;
for (k = 0; k < n; k++) used[k] = 0;
queue[0] = i; used[i] = 1;
currentVert = 0; levelVertex = 1; queueEnd = 1;
while (currentVert < queueEnd) /* cho đến khi hàng đợi trống rỗng */
{
    for (p = currentVert; p < levelVertex; p++) {
        /* p - lấy phần tử tiếp theo từ hàng đợi */
        printf("%u ", queue[p]+1);
        currentVert++;
        /* cho mỗi thừa kế bắt buộc j trong số queue[p] */
        for (j = 0; j < n; j++)
            if (A[queue[p]][j] && !used[j]) {
                queue[queueEnd++] = j;
                used[j] = 1;
            }
    }
    printf("\n");
    levelVertex = queueEnd;
}
}

int main() {
printf("Duyệt theo chiều rộng từ đỉnh %u: \n", v);
BFS(v-1);
return 0;
}

```

Kết quả thực hiện chương trình:

duyệt theo chiều rộng từ đỉnh 5:

5
2 7
1 3 4 6 10
12 11

Bài tập

- ▷ 5.3. Chương trình trên thực hiện một thuật toán để duyệt đồ thị theo chiều rộng từ một đỉnh cho trước. Trong những trường hợp nào, được thực hiện cho một đỉnh tùy ý, nó sẽ đi qua tất cả các đỉnh của đồ thị?
- ▷ 5.4. Chương trình nên được sửa đổi như thế nào để, trong trường hợp các đỉnh vẫn cần thiết, duyệt theo chiều rộng mới được thực hiện trên chúng?

5.3.2. Duyệt theo chiều sâu

Tìm kiếm chiều sâu (DFS). Tìm kiếm theo độ sâu - nó là một phần không thể thiếu của một số thuật toán phức tạp hơn là một ý tưởng cơ bản ở một trong những các phương pháp cơ bản để giải quyết các bài toán toàn diện - tìm kiếm có trả lại.

Trong khi duyệt theo chiều rộng kiểm tra các đỉnh của đồ thị một cách tuần tự theo các cấp, duyệt theo chiều sâu từ một đỉnh có xu hướng “đi xuống” sâu nhất có thể trong quá trình duyệt. Thuật toán để duyệt đồ thị theo chiều sâu được mô tả một cách đệ quy dễ dàng nhất: chúng ta bắt đầu từ đỉnh ban đầu đã chọn $i \in V$, đánh dấu nó là đã thăm và tiếp tục thu thập dữ liệu một cách đệ quy theo chiều sâu cho từng phần tử kế tiếp không được kiểm tra của nó, tức là chức năng duyệt chuyên sâu ($DFS(i)$) trông giống như sau:

- 1) Chúng ta xem xét i .
- 2) Chúng ta đánh dấu i là bỏ qua.
- 3) Đối với mỗi sự cố với i cạnh $(i, j) \in E$ Nếu j là đỉnh cần thiết của đồ thị, chúng ta thực hiện $DFS(j)$ một cách đệ quy.

Chúng ta sẽ áp dụng DFS cho ví dụ trong Hình 5.8. Trong trường hợp có nhiều hơn một cạnh là ngẫu nhiên với đỉnh được đề cập (bước 2), Chúng ta sẽ tiến hành tuần tự từ các đỉnh có số nhỏ hơn đến các đỉnh có số lượng lớn hơn. Do đó kết quả của $DFS(1)$ sẽ là: 1, 2, 3, 6, 7, 5, 10, 11, 12, 4 và của $DFS(3) : 3, 2, 1, 4, 12, 6, 7, 5, 10, 11$.

Chương trình duyệt sẽ được thực hiện trực tiếp như mô tả ở trên (thông qua hàm đệ quy $DFS(i)$). Trong mảng Boolean used[] hùng ta khởi tạo nó bằng các số không, và khi truy cập vào đỉnh i , chúng ta thực hiện phép gán used[i] = 1.

Việc nhận ra như sau:

Chương trình 5.2. Duyệt theo chiều sau(502dfs.c)

```
#include <stdio.h>
/* Số đỉnh tối đa trong cột */
#define MAXN 200
/* Số đỉnh trong cột */
const trái dấu n = 14;
/* Duyệt theo chiều sâu với đỉnh đầu v */
const trái dấu v = 5;
/* Ma trận cạnh kề của đồ thị */
const char A[MAXN][MAXN] = {
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1}
};

char used[MAXN];
/* Duyệt theo chiều sâu từ đỉnh */
void DFS(unsigned i)
{
    unsigned k;
    used[i] = 1;
    printf("%u ", i+1);
    for (k = 0; k < n; k++)
        if (A[i][k] && !used[k]) DFS(k);
}

int main() {
    unsigned k;
    for (k = 1; k < n; k++) used[k] = 0;
```

```

printf("Duyệt theo chiều sâu từ đỉnh %u: \n", v);
DFS(v-1);
printf("\n");
return 0;
}

```

Kết quả thực hiện chương trình:

duyệt độ sâu từ đỉnh 5:

5 2 1 3 6 7 10 11 12 4

Bảng 5.2 so sánh mức độ phức tạp của việc duyệt qua đồ thị trong một số kiểu biểu diễn (độ phức tạp giống nhau khi duyệt theo chiều rộng và chiều sâu). Chúng ta để nó cho người đọc để tìm hiểu xem chúng diễn ra như thế nào.

	Ma trận lân cận	Danh sách những cạnh kề	Danh sách các cạnh
Duyệt đồ thị theo DFS hoặc BFS	$\Theta(n^2)$	$\Theta(n + m)$	$\Theta(n.m)$

Bảng 5.2. Độ phức tạp của việc duyệt qua một đồ thị trong các biểu diễn khác nhau.

Trước khi kết thúc đoạn này, chúng ta sẽ xem xét ngắn gọn cách chúng ta có thể tìm thấy một cây bao phủ tùy ý trong đồ thị bằng cách sử dụng tìm kiếm theo chiều sâu (hoặc tìm kiếm theo chiều rộng, tương ứng) (xem Định nghĩa 5.14).

Gọi $G(V, E)$ là một đồ thị vô hướng liên thông. Chúng ta sẽ xây dựng cây bao phủ $T(V, D)$ theo thuật toán sau:

1) Ban đầu T là một đồ thị trong đó các cạnh không tham gia, tức là ta khởi tạo $D = \emptyset$.

2) Thực hiện duyệt theo độ sâu trong G . Đổi với mỗi lần chuyển đổi đê quy trong quá trình duyệt từ đỉnh i đến lân cận j yêu cầu của nó, chúng ta thêm cạnh (i, j) vào D .

Dễ dàng chứng minh rằng các điều kiện của định nghĩa cây bao phủ sẽ được đáp ứng: đồ thị con kết quả được kết nối (vì G được kết

nối, duyệt chiều sâu / chiều rộng sẽ "đạt đến" tất cả các đỉnh của đồ thị) và không theo chu trình (mỗi đỉnh được duyệt) được xem xét nhiều nhất một lần).

Bài tập

▷ 5.5. Chứng minh các kết quả trong Bảng 5.2.

5.4. Đường dẫn, chu trình và dòng chảy tối ưu trong đồ thị

Một trong những bài toán đồ thị phổ biến nhất là tìm đường đi tối ưu. Nếu chúng ta xem lại ví dụ 1) từ đầu chương và giải thích các đỉnh của đồ thị là các khu định cư và các đường nối chúng dưới dạng các đường phố (đường trực tiếp), một bài toán thực tế là tìm một con đường có độ dài tối thiểu giữa hai khu định cư. Ngược lại, nếu việc ghé thăm mỗi đường phố (hoặc ngôi làng) mang lại lợi nhuận nhất định, thì mục tiêu của chúng ta có thể là tìm ra con đường tối đa (tức là con đường mà chúng ta sẽ thu được nhiều nhất) giữa hai điểm. Nói chung, các điều kiện cho tính tối ưu của tuyến đường mà chúng ta sẽ tìm kiếm có thể được xác định theo nhiều cách khác nhau. Ví dụ, có thể cần phải áp dụng các hạn chế bổ sung (ví dụ: các điều khoản); được yêu cầu đến thăm từng ngôi làng chính xác một lần, hoặc băng qua từng con phố chính xác một lần, v.v. Tuy nhiên, cần xác định rõ hai điều:

- *cách ước tính một tuyến đường* (ví dụ - dưới dạng tổng trọng lượng của các cạnh liên quan đến nó)
- *tiêu chí tối ưu của lộ trình* (ví dụ: chúng ta muốn giảm thiểu số tổng trên)

Như một sự tiếp tục tự nhiên của việc tìm kiếm các đường đi tối ưu trong một đồ thị, chúng ta sẽ xem xét việc tìm kiếm các chu trình (tối ưu) trong một đồ thị (nhớ lại rằng một chu trình là một đường mà đỉnh đầu và đỉnh cuối cùng trùng nhau). Có một số kiểu chu trình đáng chú ý mà chúng ta sẽ đặc biệt chú ý.

Chúng ta sẽ kết thúc đoạn này với bài toán tìm dòng chảy cực đại trong đồ thị - một bài toán cơ bản được tìm thấy ứng dụng trong một số lượng lớn các bài toán thực tế.

5.4.1. Các ứng dụng trực tiếp của thuật toán duyệt

- đường đi ngắn nhất giữa hai đỉnh theo số đỉnh

Cho đồ thị $G(V, E)$ và hai đỉnh i và j của nó. Chúng ta tìm một đường đi trong G có đầu là đỉnh i và cuối là đỉnh j , có độ dài tối thiểu trong số các đỉnh tham gia vào nó. Sau khi thực hiện duyệt theo chiều rộng, chúng ta có thể dễ dàng biên dịch một thuật toán để tìm một đường đi tối thiểu như vậy: Chúng ta chạy $BFS(i)$ và nếu ở bất kỳ bước nào chúng ta đạt đến j , nó sẽ ngay lập tức theo sau rằng có một đường đi giữa hai đỉnh, đồng thời chúng ta cũng có một con đường tối thiểu cụ thể.

Để làm ví dụ, chúng ta sẽ xem xét đồ thị trong Hình 5.8. Hãy tìm đường đi từ đỉnh 1 đến đỉnh 10. Sau đó, kết quả của việc thực hiện $BFS(1)$ sẽ là:

cấp độ 1: 1

cấp độ 2: 2

cấp độ 3: 3, 4, 5

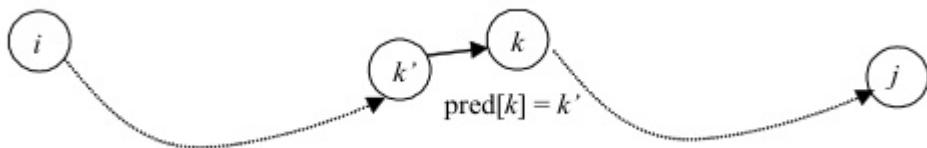
cấp độ 4: 7, 6, 12

cấp độ 5: 10

Khi chúng ta đã đạt đến đỉnh cuối cùng j (trong trường hợp này là $j = 10$), rõ ràng độ dài của đường đi nhỏ nhất là mức mà chúng ta đã đi qua j , trừ đi một. Tuy nhiên, làm thế nào để tìm và in các đỉnh trung gian tham gia vào đường? Vẫn đề dễ dàng được giải quyết nếu ở mỗi cấp độ chúng ta có tiền thân của đỉnh được duyệt, tức là đỉnh từ lần duyệt lặp lại trước đó, từ đó chúng ta thêm đỉnh hiện tại làm hàng xóm ngay lập tức. Với mục đích này, chúng ta sẽ nhập mảng $\text{pred}[\text{i}]$: trong đó, chúng ta sẽ giữ các giá trị trước của mỗi đỉnh bỏ qua. Đối với đỉnh ban đầu i , $\text{pred}[i]$ chúng ta sẽ khởi tạo với giá trị -1 . Do đó, $\text{pred}[k]$ sẽ chứa đỉnh mà từ đó chúng ta đã chuyển đến k .

Việc phục hồi các đỉnh trung gian tham gia vào đường đi nhỏ nhất từ i đến j được thực hiện theo sơ đồ sau:

```
while ( $j \neq i$ ) {
    print( $j$ );
     $j = \text{pred}[j]$ ;
```



Hình 5.9. Tiết kiệm một đường đi.

```

}
print(j);

```

Đoạn mã giả trên sẽ in ra đường dẫn tối thiểu theo thứ tự ngược lại (tức là từ j đến i). Để có được đường dẫn một cách chính xác, chúng ta có thể viết các đỉnh trong một mảng, sau đó chúng ta có thể in theo thứ tự ngược lại. Do đó, chúng ta hoàn toàn sử dụng một ngăn xếp - trước tiên hãy thêm tất cả các phần tử theo thứ tự, sau đó tắt và in. Cách sau là một cách cổ điển để "đảo ngược kết quả" và được thực hiện một cách thanh lịch nhất với đệ quy (xem cách triển khai bên dưới - hàm printPath). Độ phức tạp của thuật toán là $\Theta(n + m)$. Triển khai đầy đủ như sau:

Chương trình 5.3. Đường đi ngắn nhất(503bfsminw.c)

```

#include <stdio.h>
#define MAXN 200 /* Số đỉnh tối đa trong đồ thị*/
/* Số đỉnh trong đồ thị */
const unsigned n = 14;
const unsigned sv = 1; /* Đỉnh bắt đầu */
const unsigned ev = 10; /*Đỉnh kết thúc */
/* Ma trận kề của đồ thị */
const char A[MAXN][MAXN] = {
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
{0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0},
{0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1},

```

```

{0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
{0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0}
};

int pred[MAXN];
char used[MAXN];

/* Duyệt chiều rộng của đỉnh trong khi giữ nguyên giá trị trước*/
void BFS(unsigned i)
{
    unsigned queue[MAXN];
    unsigned currentVert, levelVertex, queueEnd, k, p, j;
    for (k = 0; k < n; k++) queue[k] = 0;
    queue[0] = i; used[i] = 1;
    currentVert = 0; levelVertex = 1; queueEnd = 1;
    while (currentVert < queueEnd) { /* Khi hàng đợi khác rỗng */
        for (p = currentVert; p < levelVertex; p++) {
            /* p - lấy phần tử tiếp theo từ hàng đợi */
            currentVert++;
            /* với mỗi đỉnh kè bắt buộc j trong số queue[p] */
            for (j = 0; j < n; j++) {
                if (A[queue[p]][j] && !used[j]) {
                    queue[queueEnd++] = j;
                    used[j] = 1;
                    pred[j] = queue[p];
                }
            }
            levelVertex = queueEnd;
        }
    }

    /* In các đỉnh của đường đi nhỏ nhất và trả về độ dài của nó*/
    unsigned printPath(unsigned j)
    {
        unsigned count = 1;
        if (pred[j] > -1) count += printPath(pred[j]);
        printf("%u ", j + 1); /* In một đỉnh khác của đường dẫn được tìm
        thấy */
        return count;
    }
}

```

```

    }

void solve(unsigned start, unsigned end)
{ unsigned k;
  for (k = 0; k < n; k++) { used[k] = 0; pred[k] = -1; }
  BFS(start);
  if (pred[end] > -1) {
    printf("Đường được tìm thấy: \n");
    printf("\nĐộ dài của đường là %u.\n", printPath(end));
  }
  else
    printf("Không có đường đi giữa hai đỉnh! \n");
}

int main()
{
  solve(sv-1, ev-1);
  return 0;
}

```

Kết quả thực hiện chương trình:

Đường dẫn được tìm thấy là:

1 2 5 7 10

Con đường là 5.

Bài tập

▷ 5.6. Sửa đổi chức năng duyệt BFS() để nó ngừng hoạt động khi đạt đến mức cao nhất cuối cùng.

- Kiểm tra xem đồ thị có tuần hoàn không

Sử dụng thuật toán thu thập thông tin chuyên sâu, chúng ta sẽ xây dựng một thuật toán để kiểm tra xem một đồ thị vô hướng có phải là theo chu trình hay không. Thuật toán để kiểm tra tính chu trình bao gồm những điều sau: Thực hiện thu thập thông tin sâu. Nếu tại bất kỳ bước nào của quá trình thu thập thông tin, nó chỉ ra rằng đỉnh i được đề cập có một hàng xóm mà chúng ta đã đi qua và khác với đỉnh i trước đó, thì đồ thị đó có chứa một vòng lặp.

Chúng ta sẽ thực hiện thuật toán bằng cách sửa đổi hàm DFS() (vì tham số của nó, chúng ta sẽ thêm **int** parent - hàm mẹ ở trên cùng

mà chúng ta hiện đang ở). Nếu chúng ta đạt từ đỉnh i hiện tại đến đỉnh mà chúng ta đã ở đó, không phải là giá trị gốc, thì vòng lặp đã đóng. Chúng ta sẽ đánh dấu các đỉnh đã thăm như bình thường trong $\text{used}[]$ đã sử dụng, và trong hàm chính, chúng ta sẽ gọi $\text{DFS}(i, -1)$ cho mỗi đỉnh i , điều này vẫn chưa được xem xét cho đến nay. Giá trị -1 cho gốc có nghĩa là đỉnh này là đỉnh đầu tiên thu thập thông tin thành phần kết nối tương ứng và do đó không có giá trị chính.

Chương trình 5.4. Đường đi ngắn nhất(504formcycl.c)

```
#include <stdio.h>
/* Số đỉnh tối đa trong đồ thị */
#define MAXN 200
/* Số đỉnh trong đồ thị */
const unsigned n = 14;
/* Ma trận kè của đồ thị */
const char A[MAXN][MAXN] = {
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
{0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0},
{0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0},
{0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0},
{0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
};

char used[MAXN], cycl;
```

/* Sửa đổi DFS */

```
void DFS(unsigned i, int parent)
{ unsigned k;
    used[i] = 1;
    for (k = 0; k < n; k++) {
        if (cycl) return;
        if (A[i][k]) {
```

```

if (used[k] && k != parent) {
    printf("Đồ thị có tính chu trình! \n");
    cycl = 1;
    return;
}
else if (k != parent)
    DFS(k, i);
}
}

int main() {
    unsigned k, i;
    for (k = 0; k < n; k++) used[k] = 0;
    cycl = 0;
    for (i = 0; i < n; i++) {
        if (!used[i]) DFS(i, -1);
        if (cycl) break;
    }
    if (0 == cycl)
        printf("đồ thị là một cây (không chứa vòng lặp)! \n");
    return 0;
}

```

Kết quả thực hiện chương trình:

Đồ thị là tuần hoàn!

Độ phức tạp của thuật toán là $\Theta(n + m)$, và khi biểu diễn đồ thị bằng ma trận lân cận, như trong cách triển khai được đề xuất, nó là $\Theta(n^2)$.

Bài tập

- ▷ 5.7. Chương trình trên có hoạt động trong trường hợp đồ thị có định hướng không?
- ▷ 5.8. Bạn có thể thực hiện những sửa đổi nào trong trường hợp đồ thị có định hướng? Mã được đơn giản hóa hay phức tạp?
 - Tìm tất cả các đường đi đơn giản giữa hai đỉnh

Trong phần này, chúng ta sẽ xem xét thuật toán thu thập thông tin chuyên sâu. Trong *DFS* từ phần 5.3.2, khi đỉnh mà chúng ta đang xem xét hiện có nhiều hơn một người kế nhiệm, chúng ta đã chọn tiếp tục với đỉnh có số thấp nhất. Việc thu thập thông tin sẽ đúng nếu chúng ta chọn tiếp tục với đỉnh có số lượng lớn nhất, cũng như theo bất kỳ cách sắp xếp nào khác của các đỉnh. Chúng ta sẽ thấy trong giây lát quan sát này sẽ giúp chúng ta xây dựng một thuật toán tìm tất cả các đường đi có thể có giữa hai đỉnh như thế nào.

Khi tìm kiếm tất cả các đường đi từ i đến j , chúng ta phải áp dụng tính hết hoàn toàn (có thể thực hiện bằng cách quay lại tìm kiếm - xem Chương ??). Do đó, độ phức tạp của thuật toán theo sau sẽ là cấp số nhân. Một lần nữa, chúng ta sẽ lưu ý rằng chúng ta đang tìm kiếm tất cả các con đường đơn giản, tức là không chứa các đỉnh lặp lại. Ngược lại, nếu đường dẫn chứa đỉnh lặp lại, điều đó có nghĩa là nó có chứa một vòng lặp - khi đó bạn sẽ có thể tìm thấy nhiều đường dẫn xuất phát từ đỉnh đã cho (mỗi đường tiếp theo sẽ có được bằng cách "quay" một hoặc nhiều lần trong chu trình).

Thuật toán tìm tất cả các đường dẫn đơn giản

Nếu chúng ta thực hiện thu thập thông tin quen thuộc theo chiều rộng hoặc chiều sâu, bắt đầu từ i , chúng ta sẽ tìm thấy một đường đi đơn giản có thể từ i đến j , vì chúng ta nhất thiết phải đi qua tất cả các đỉnh của cùng một thành phần (bao gồm cả đỉnh j). Tất nhiên, nếu hai đỉnh không thuộc cùng một thành phần của kết nối, thì không có đường đi nào giữa chúng và như vậy sẽ không được tìm thấy. Việc sửa đổi thuật toán thu thập thông tin theo chiều sâu bao gồm thực hiện đệ quy tuần tự *DFS*(k) cho mỗi đỉnh k liền kề với đỉnh i hiện đang được xem xét và không chỉ cho lân cận có số lượng tối thiểu. Bằng cách này, chương trình chính trong chương trình sẽ không thay đổi:

```
for (k = 0; k < n; k++)
    if (A[i][k] && !used[k]) allDFS(k, j);
```

Sự khác biệt sẽ là trong cách triển khai mới, chúng ta sẽ thực hiện phép gán *used[i] = 0* sau khi trở về từ đệ quy, trong khi trong *DFS* thông thường thì điều này không đúng như vậy. Trong ví dụ của Hình 5.8 điều này có nghĩa như sau: Bây giờ chúng ta hãy nhìn vào

đỉnh 2 - nó có bốn người hàng xóm: 1, 4, 3, 5. Chúng ta đến từ đỉnh 1 và do đó chúng ta sẽ không tiếp tục nó. Thực hiện **for**-chu trình, chọn đỉnh 3 và bắt đầu *DFS*(3). Do đó, chúng ta sẽ tiếp tục thu thập dữ liệu một cách đệ quy ngoài đỉnh 3, nhưng sau khi quay trở lại từ đệ quy ở cuối *DFS*(3), phép gán used[3] = 0 sẽ được thực hiện. Hơn nữa, khi quá trình thu thập thông tin tiếp tục với các hàng xóm tiếp theo - đỉnh 4 và 5, trong những lần thu thập thông tin này, nó sẽ có thể vượt qua đỉnh 3. Đây không phải là trường hợp với *DFS* thông thường - với nó, chúng ta đã từng đánh dấu đỉnh 3 với used[3] = 1, nó không còn khả dụng ở bất kỳ bước thu thập thông tin tiếp theo nào.

Chúng ta cũng sẽ giới thiệu một đường dẫn mảng path[], trong đó chúng ta sẽ giữ các đỉnh theo thứ tự của đường đi ngang của chúng. Do đó, nếu tại một số bước chúng ta đạt đến mục tiêu đỉnh cuối cùng j, chúng ta có thể in ra đường dẫn hiện tại. Sau đây là cách triển khai hàm allDFS (i, j). Nó được đáp ứng bởi hai tham số: đỉnh mà chúng ta bắt đầu từ i, và đỉnh mà chúng ta phải đạt tới j. Tất cả các đầu vào được đặt ở đầu chương trình dưới dạng hằng số.

Chương trình 5.5. Tìm tất cả đường dẫn(505btdfs.c)

```
#include <stdio.h>
/* Số đỉnh tối đa trong đồ thị */
#define MAXN 200
/* Số đỉnh trong đồ thị*/
const unsigned n = 14;
const unsigned sv = 1; /* Đỉnh xuất phát */
const unsigned ev = 10; /* Đỉnh kết thúc */
/* Ma trận kề của đồ thị */
const char A[MAXN][MAXN] = {
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
{0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},
{0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
{0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0},
{0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1},
{0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0},
```

```

{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},  

{0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0},  

{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1},  

{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0}  

};  
  

char used[MAXN];  

unsigned path[MAXN], count;  
  

void printPath(void)  

{ unsigned k;  

for (k = 0; k <= count; k++)  

printf("%u ", path[k] + 1);  

printf("\n");  

}  
  

/* Tìm tất cả các đường đi đơn giản giữa các đỉnh i và j */  

void allDFS(unsigned i, unsigned j)  

{ unsigned k;  

if (i == j) {  

path[count] = j;  

printPath();  

return;  

}  
  

/* đánh dấu đỉnh đã ghé thăm */  

used[i] = 1;  

path[count++] = i;  

for (k = 0; k < n; k++) /* đệ quy cho tất cả những đỉnh kề của i */  

if (A[i][k] && !used[k]) allDFS(k, j);  

/* return: bỏ đánh dấu đỉnh đã ghé thăm */  

used[i] = 0; count--;  

}  
  

int main() {
unsigned k;
for (k = 0; k < n; k++) used[k] = 0;
count = 0;
printf("Những con đường đơn giản ở giữa %u và %u: \n", sv, ev);
allDFS(sv-1, ev-1);
}

```

```

    return 0;
}

```

Kết quả từ việc thực hiện chương trình:

Các đường dẫn đơn giản từ 1 đến 10:

1 2 3 6 7 10

1 2 4 12 6 7 10

1 2 5 7 10

Bài tập

- ▷ 5.9. Để triển khai một thuật toán để tìm tất cả các đường dẫn đơn giản bằng cách thu thập thông tin theo chiều rộng (BFS). Mã được đơn giản hóa hay phức tạp? Độ phức tạp của thuật toán có thay đổi không?

5.4.2. Đường tối ưu trong đồ thị

Định nghĩa 5.15. Cho được một đồ thị có hướng trọng số $G(V, E)$ với các cạnh của các cạnh cho trước là các số thực. Chiều dài của một đường trong G là tổng trọng số của các cạnh có liên quan.

Trong phần này, chúng ta sẽ tập trung vào các thuật toán để tìm độ dài đường dẫn tối đa và nhỏ nhất trong một đồ thị. Trong 5.4.1 chúng ta đã xem xét một thuật toán để tìm tất cả các đường đi giữa hai đỉnh. Một giải pháp như vậy với sự cạn kiệt hoàn toàn là cách duy nhất để giải quyết một số vấn đề về tìm đường đi tối ưu, nhưng đặc biệt trong trường hợp chúng ta đang tìm đường đi tối đa hoặc tối thiểu dưới dạng tổng / tích của trọng lượng của các cạnh cấu thành của nó, có các thuật toán hiệu quả hơn nhiều.

Trong một số tác vụ, có thể xác định độ dài đường dẫn không phải là một tổng, mà là một số hàm khác của trọng số của các cạnh (và thậm chí cả các đỉnh) liên quan đến đường dẫn. Để các thuật toán này cũng vẫn có giá trị trong những trường hợp này, phải có một số tiêu chí cụ thể về tính tối ưu (chúng phụ thuộc vào thuật toán cụ thể đang được xem xét).

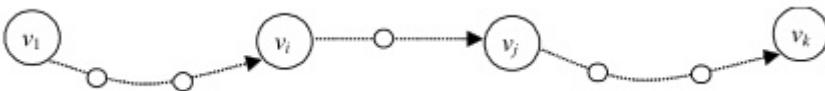
Trước khi chuyển sang phần thực, chúng ta sẽ chú ý đến một số chi tiết chưa rõ ràng. Vì có thể không có hạn chế nào đối với đường

dẫn là đơn giản, chúng ta phải cẩn thận trong trường hợp đồ thị có chứa một vòng lặp. Ví dụ: nếu chúng ta tìm kiếm một đường đi nhỏ nhất và có một chu trình âm (chu trình có độ dài âm), chúng ta sẽ có thể "xoay" trên chu trình này bất kỳ số lần nào, trong đó chiều dài của đường dẫn (bằng tổng của các cạnh trong đó) sẽ giảm nhiều tùy ý đến trừ vô cùng. Tương tự, nếu chúng ta tìm một đường đi cực đại và có một chu trình dương, thì với mỗi đường đi chứa nó, chúng ta sẽ có thể "xoay" chu trình, do đó thu được một chiều dài lớn tùy ý.

Các thuật toán mà chúng ta sẽ bắt đầu sẽ là tìm đường đi nhỏ nhất trong đồ thị. Nếu chúng ta tìm kiếm đường dẫn tối đa, chúng ta có thể dễ dàng sửa đổi chúng. Một giải pháp khả thi là "đảo ngược" trọng số của các cạnh của đồ thị: chúng ta sẽ thay đổi trọng số $f(i, j)$ của mỗi cạnh (i, j) thành $-f(i, j)$. Nếu sau đó chúng ta thực hiện một thuật toán để tìm đường đi nhỏ nhất trong đồ thị mới thu được, thì đường đi tìm được sẽ là đường lớn nhất cho đường ban đầu. Tuy nhiên, cách tiếp cận này không phải lúc nào cũng có thể áp dụng và không phổ biến để tìm các loại đường dẫn cực đại khác nhau trong các loại đồ thị khác nhau.

Hầu hết các thuật toán để tìm một đường cực trị đều dựa trên dấu hiệu tối ưu sau:

Định lý 5.1 (dấu hiệu của tính tối ưu). *Giả sử một đồ thị, một cặp đỉnh của nó (v_1, v_k) và một đường đi nhỏ nhất $p = (v_1, v_2, \dots, v_k)$ giữa v_1 và v_k . Khi đó, với $1 \leq i < j \leq k$, đường đi $p' = (v_i, v_{i+1}, \dots, v_j)$ là đường đi nhỏ nhất từ đỉnh v_i đến v_j .*



Hình 5.10. Tối ưu hóa đường.

Tính hợp lệ của định lý cuối cùng có thể dễ dàng chứng minh bằng cách giả thiết ngược lại (Chúng ta để độc giả xem như một bài tập).

Để tìm những con đường tối thiểu, một mảng thường được giới thiệu nhất, các giá trị của nó có nghĩa là giới hạn trên của những con đường tối thiểu bắt buộc. Ở mỗi bước, một trong các giới hạn trên được giảm xuống (cách thực hiện điều này phụ thuộc vào thuật toán cụ thể) cho đến khi đạt đến mức tối thiểu cần thiết.

Chúng ta sẽ chứng minh sơ đồ tổng quát được coi là với một số thuật toán nổi tiếng nhất để tìm đường dẫn tối thiểu trong đồ thị.

Bài tập

▷ 5.10. Hãy chứng minh dấu hiệu của sự tối ưu.

- Bất đẳng thức của tam giác

Các thuật toán của Ford-Bellman, Floyd và Dijkstra dựa trên bất đẳng thức của một tam giác: tổng độ dài của hai cạnh bất kỳ lớn hơn độ dài của cạnh thứ ba. Gọi $d(i, j)$ là khoảng cách giữa các đỉnh i và j . Khi đó, trong trường hợp đồ thị có hướng, các bất đẳng thức của tam giác đối với bộ ba đỉnh i, j, k tùy ý có thể được viết như sau:

$$\begin{aligned} d(i, k) + d(k, j) &\geq d(i, j) \\ d(i, j) + d(j, k) &\geq d(i, k) \\ d(k, i) + d(i, j) &\geq d(k, j) \\ d(j, k) + d(k, i) &\geq d(j, i) \\ d(k, j) + d(j, i) &\geq d(k, i) \\ d(j, i) + d(i, k) &\geq d(j, k) \end{aligned}$$

Trong trường hợp đồ thị vô hướng, chúng ta có $d(x, y) = d(y, x)$ cho mỗi cặp đỉnh x, y và ba bất đẳng thức cuối cùng trở thành dư thừa. Mỗi thuật toán được liệt kê dựa trên việc kiểm tra tuần tự (một số) bất đẳng thức của tam giác và trong trường hợp vi phạm, thiết lập đẳng thức.

Bài tập

▷ 5.11. Chứng minh sự cần thiết phải quan sát bất đẳng thức của tam giác như là một hệ quả của dấu hiệu của tối ưu.

- Thuật toán Ford-Bellman

Cho là một đồ thị có hướng $G(V, E)$ và tìm khoảng cách ngắn nhất từ một đỉnh s đến tất cả các đỉnh khác. Chúng ta sẽ giả sử rằng chúng ta đang làm việc với ma trận trọng số $A[i][j]$ của đồ thị G . Trong trường hợp cạnh (i, j) không tồn tại, giá trị của $A[i][j]$ sẽ là $+\infty$ (và vì chúng ta đang làm việc với cấu trúc dữ liệu hữu hạn, đây sẽ là giá trị cho phép lớn nhất của kiểu các phần tử của A).

Thuật toán Ford-Bellman:

- 1) Chúng ta nhập một mảng $D[]$, và sau khi hoàn thành thuật toán, $D[i]$ sẽ chứa độ dài của đường đi nhỏ nhất từ s đến mọi đỉnh i khác của đồ thị. Chúng ta khởi tạo $D[i] = A[s][i]$, cho mỗi đỉnh $i \in V$.
- 2) Chúng ta cố gắng tối ưu hóa giá trị của $D[i]$ cho mỗi $i \in V$ theo cách sau: Với mỗi $j \in V$, nếu $D[i] > D[j] + A[j][i]$, chúng ta gán $D[i] = D[j] + A[j][i]$.
- 3) Sau khi lặp lại bước 2) $n - 2$ lần, mảng $D[]$ sẽ chứa các đường đi nhỏ nhất cần thiết.

Chương trình 5.6. Thuật toán Ford-Bellman (506belman.c)

```
for (k = 1; k <= n - 2; k++) /* lặp lại (n-2) lần */
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if (D[i] > D[j] + A[j][i])
                D[i] = D[j] + A[j][i];
```

Rõ ràng, độ phức tạp của thuật toán là $\Theta(n^3)$, vì bước 2) có độ phức tạp $\Theta(n^2)$ và được thực hiện $n - 2$ lần. Điều duy nhất cần cẩn thận trong trường hợp này theo quan điểm của hiện thực là tính tổng $D[j] + A[j][i]$ để không bị tràn (khi bất kỳ $D[j]$ hoặc $A[j][i]$ được khởi tạo với giá trị lớn nhất cho kiểu).

Thuật toán Ford-Bellman hợp lệ cho các đồ thị có trọng số cạnh tùy ý: cả dương và âm. Tất nhiên, trong sự hiện diện của một chu trình âm, chúng ta không thể nói về một con đường tối thiểu (xem ở trên). Một tính năng hữu ích của thuật toán Ford-Bellman là nó cho phép phát hiện các vòng lặp âm: Nếu sau khi hoàn thành, đối với bất kỳ cặp đỉnh (i, j) nào được thỏa mãn $D[i] > D[j] + A[j][i]$, thì đồ

thị chứa một vòng lặp âm (Tại sao?).

Bài tập

- ▷ 5.12. Hãy chứng minh thuật toán Ford-Bellman.
- ▷ 5.13. Tại sao chu trình ngoài cùng của thuật toán trên được lặp lại $n - 2$ lần mà không phải ví dụ như n hay $n - 1$?
- ▷ 5.14. Chứng minh rằng nếu sau khi hoàn thành thuật toán Ford-Bellman cho bất kỳ cặp đỉnh nào (i, j) , $D[i] > D[j] + A[j][i]$ được thỏa mãn, thì đồ thị chứa một số âm xe đạp.
- ▷ 5.15. Phát biểu ngược lại có đúng không, cụ thể là: Nếu đồ thị chứa chu trình âm thì sau khi hoàn thành thuật toán Ford-Bellman sẽ có một cặp đỉnh (i, j) mà $D[i] > D[j] + A[j][i]$?

- Thuật toán của Floyd

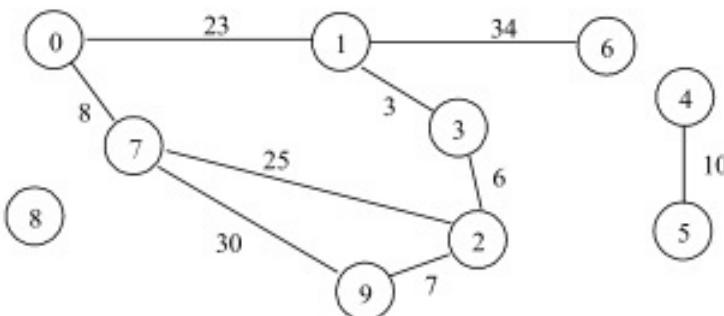
Thuật toán của Floyd tương tự như thuật toán của Ford-Bellman. Sự khác biệt cơ bản là khi nó kết thúc, chúng ta có độ dài của các đường đi nhỏ nhất giữa mỗi cặp đỉnh trong đồ thị mà không cần thêm bộ nhớ (sử dụng ma trận trọng số của nó). Điều này có nghĩa là nếu ngay từ đầu trong $A[i][j]$ trọng số của cạnh (i, j) được viết, thì sau khi thực hiện thuật toán Floyd trên ma trận $A[|V|]$, giá trị của $A[i][j]$ sẽ là độ dài của đường đi nhỏ nhất giữa i và j .

Thuật toán bao gồm những điều sau: với mỗi hai đỉnh $i, j \in V$, $A[i][j]$ được gán nhỏ hơn $A[i][j]$ và $A[i][k] + A[k][j]$, đối với mỗi $k \in V$ đỉnh. Ý nghĩa của đỉnh k , với tư cách là đỉnh trung gian để tối ưu hóa đường, sẽ trở nên rõ ràng hơn trong phần trình bày của đoạn tiếp theo (lược đồ tổng quát của Floyd).

Độ phức tạp của thuật toán Floyd là $\Theta(n^3)$. Như với Ford-Bellman, sự hiện diện của các cạnh âm không phải là vấn đề, miễn là đồ thị không chứa các vòng âm.

Chúng ta sẽ thực hiện kiểm tra ở trên với ba chu trình lồng nhau - k, i và j . Để thuận tiện, khi so sánh bước chính, thay vì 0, chúng ta sẽ sử dụng giá trị MAX_VALUE, nghĩa là không có cạnh trong ma trận trọng số (trong thực tế, số 0 có thể là giá trị trọng số hợp lệ).

Để chương trình hoạt động chính xác, MAX_VALUE phải có giá trị lớn hơn $n \cdot d_{\max}$, trong đó d_{\max} là trọng số lớn nhất của một cạnh của đồ thị (yêu cầu này có thể được làm yếu đi: chỉ cần lấy tổng các cạnh dương là đủ). Cũng nên cẩn thận để làm tràn phần tổng kết: Vì chúng ta có hai phép cộng, MAX_VALUE không được vượt quá MAXINT/2, trong đó MAXINT là giá trị lớn nhất cho kiểu int trong trình biên dịch C tương ứng (trong hầu hết các môi trường, hằng số MAXINT được xác định trong tiêu đề tệp <values.h>).



Hình 5.11. Tối ưu hóa đường.

Sau đây là mã nguồn của chương trình. Dữ liệu đầu vào tương ứng với đồ thị không định hướng trong Hình 5.11 (Tất nhiên, thuật toán của Floyd cũng như chương trình bên dưới sẽ hoạt động chính xác với đồ thị có định hướng). Khi đặt dữ liệu đầu vào là hằng số, chúng ta đã sử dụng giá trị 0 để chỉ ra sự không có cạnh trong ma trận trọng số (để rõ ràng hơn trong mã nguồn). Sau đó (ở đầu hàm floyd() chính) tất cả các trường 0 đều được gán giá trị MAX_VALUE

Chương trình 5.7. Thuật toán floyd (507floyd.c)

```
#include <stdio.h>
/* Số đỉnh tối đa trong đồ thị */
#define MAXN 150
#define MAX_VALUE 10000
/* Số đỉnh trong đồ thị */
const unsigned n = 10;
/* Ma trận trọng số của đồ thị */
const unsigned A[MAXN][MAXN] = {
{ 0, 23, 0, 0, 0, 0, 0, 8, 0, 0 },
```

```

{23, 0, 0, 3, 0, 0, 34, 0, 0, 0 },
{ 0, 0, 0, 6, 0, 0, 0, 25, 0, 7 },
{ 0, 3, 6, 0, 0, 0, 0, 0, 0, 0 },
{ 0, 0, 0, 0, 0, 10, 0, 0, 0, 0 },
{ 0, 0, 0, 0, 10, 0, 0, 0, 0, 0 },
{ 0, 34, 0, 0, 0, 0, 0, 0, 0, 0 },
{ 8, 0, 25, 0, 0, 0, 0, 0, 0, 30 },
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
{ 0, 0, 7, 0, 0, 0, 0, 30, 0, 0 }
};

/* Tìm độ dài của đường đi nhỏ nhất giữa mỗi cặp đỉnh*/
void floyd(void)
{
    unsigned i, j, k;

    /* các giá trị 0 thay đổi thành MAX_VALUE */
    for (i = 0; i < n; i++) for (j = 0; j < n; j++)
        if (0 == A[i][j]) A[i][j] = MAX_VALUE;

    /* Thuật toán Floyd */
    for (k = 0; k < n; k++)
        for (i = 0; i < n; i++)
            for (j = 0; j < n; j++)
                if (A[i][j] > (A[i][k] + A[k][j]))
                    A[i][j] = A[i][k] + A[k][j];

    for (i = 0; i < n; i++)
        A[i][i] = 0;
}

void printMinPaths(void)
{
    unsigned i, j;
    printf("Ma trận trọng số sau khi thực hiện Floyd:\n");
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++)
            printf("%3d ", (MAX_VALUE == A[i][j]) ? 0 : A[i][j]);
        printf("\n");
    }
}

```

```
int main() {
    floyd();
    printMinPaths();
    return 0;
}
```

Kết quả thực hiện thuật toán:

Ma trận trong số sau khi thực hiện Floyd:

```
0 23 32 26 0 0 57 8 0 38
23 0 9 3 0 0 34 31 0 16
32 9 0 6 0 0 43 25 0 7
26 3 6 0 0 0 37 31 0 13
0 0 0 0 0 10 0 0 0 0
0 0 0 0 10 0 0 0 0 0
57 34 43 37 0 0 0 65 0 50
8 31 25 31 0 0 65 0 0 30
0 0 0 0 0 0 0 0 0 0
38 16 7 13 0 0 50 30 0 0
```

Chúng ta sẽ lưu ý rằng thứ tự của các chu trình là chủ yếu: chu trình của k phải ở ngoài cùng. Ý tưởng là đầu tiên chúng ta tìm các đường đi ngắn nhất với các đỉnh trung gian 1, sau đó là những con đường có các đỉnh trung gian 1 và 2, sau đó là những con đường có các đỉnh trung gian 1, 2 và 3, v.v. Rõ ràng là vị trí tương hỗ của các chu trình trên i và j là không có ý nghĩa. cả phép lệnh k, i, j và k, j, i đều đúng. Nhưng mọi quy định khác đều dẫn chúng ta đến một thuật toán không chính xác. Hãy coi ma trận như một ví dụ (lưu ý rằng nó không đối xứng, tức là tương ứng với một đồ thị có định hướng):

```
const n = 4;
int A[n][n] =
{  
// A B C D
{ 0, 0, 6, 2}, // A
{ 0, 0, 0, 0}, // B
{ 0, 1, 0, 0}, // C
{ 0, 0, 2, 0} // D
};
```

Với sự sắp xếp đúng, chúng ta thu được ma trận các đường dẫn tối

thiểu sau:

0	5	4	2
0	0	0	0
0	1	0	0
0	3	2	0

Tuy nhiên, theo thứ tự của các chu trình i, k, j ta có:

0	7	4	2
0	0	0	0
0	1	0	0
0	3	2	0

Trong trường hợp đầu tiên, đường đi ngắn nhất từ A đến B là (A, D, C, B) với độ dài là 5, và trong trường hợp thứ hai - chúng ta nhận được sai (A, C, B) với độ dài là 7. Chúng ta để người đọc quyết định tại sao nó lại xảy ra sự khác biệt này.

Lưu ý: Việc khôi phục các đường dẫn nhỏ nhất cụ thể, không chỉ tìm độ dài của chúng, sẽ được thảo luận bên dưới khi xem xét thuật toán Dijkstra. Người đọc có thể trực tiếp "chuyển" và áp dụng sơ đồ này cho thuật toán của Floyd.

Bài tập

- ▷ 5.16. Trên một cặp đỉnh (i, j) hãy khôi phục một đường đi nhỏ nhất cụ thể.
- ▷ 5.17. Trên một cặp đỉnh nhất định (i, j) hãy khôi phục tất cả các đường tối thiểu.

- Thuật toán tổng quát của Floyd

Thuật toán Floyd để tìm đường đi nhỏ nhất giữa tất cả các cặp đỉnh có thể được tóm tắt thành sơ đồ sau để tìm đường đi tối ưu:

$$\varphi^{(k)}(i, j) = F\{\varphi^{(k-1)}(i, j), \varphi^{(k-1)}(i, k) \oplus \varphi^{(k-1)}(k, j)\},$$

trong đó $\varphi^{(1)}(i, j) = f(i, j)$.

$\varphi^{(k)}$ là một hàm đại diện cho đường đi tối ưu giữa hai đỉnh, chứa không quá k đỉnh, có thể tối ưu hóa và \oplus là một phép toán hai đối

số tính toán tổng giá trị của đường đi có được bằng cách ghép hai non -đường dẫn kết hợp. Biểu diễn đệ quy như vậy cho ta ẩn tượng rằng chúng ta đang làm việc với n ma trận số: vì ma trận với các giá trị $\varphi^{(1)}(i, j)$ là ma trận trọng số của đồ thị và $\varphi^{(k)}(i, j)$ là ma trận với các đường dẫn tối ưu cần thiết. im lặng. Lược đồ trên được thực hiện lặp đi lặp lại với bộ nhớ bậc hai, như trong thuật toán của Floyd ở đoạn trước. Hàm F được sử dụng để tối ưu hóa giá trị thu được bằng phép toán \oplus .

Độ phức tạp của thuật toán tổng quát Floyd, nếu chúng ta giả định rằng độ phức tạp của việc thực hiện phép toán \oplus và tính hàm F là hằng số, giống như tiêu chuẩn: $\Theta(n^3)$.

Hãy xem cách chúng ta có thể áp dụng sơ đồ trên để tìm độ dài tối thiểu của các con đường giữa tất cả các cặp đỉnh. Trong trường hợp này, phép toán \oplus sẽ là tổng độ dài của hai đường đi. Vì chúng ta đang tìm một đường đi có độ dài nhỏ nhất, F sẽ là hàm nhỏ nhất.

Chúng ta sẽ sử dụng một ma trận đơn $A[i][j]$ để lưu kết quả của các giá trị thứ k của hàm $\varphi^{(k)}(i, j)$. Lúc đầu ta khởi tạo $A[i][j]$ với các trọng số cho trước của các cạnh của đồ thị (tức là $A[i][j]$ là ma trận các trọng số). Khi F là cực đại và không tồn tại cạnh (i, j) thì ma trận $A[i][j]$ được viết bằng 0. Giá trị của $A[i][j]$ cho các vòng lặp sẽ là + vô cực (MAX_VALUE). Ngược lại, khi F là cực tiểu và không tồn tại cạnh (i, j) thì ma trận chứa MAX_VALUE và giá trị của $A[i][j]$ là 0.

Thay vì $A[i][j] == \text{MAX_VALUE}$ hoặc $A[i][j] == 0$, thích hợp hơn là thực hiện kiểm tra sau:

if ($i \neq j$) **&&** ($i \neq k$) **&&** ($k \neq j$) { ... }

Điều này tránh việc sử dụng và khởi tạo các phần tử đường chéo $A[i][i]$.

Chúng ta sẽ minh họa những gì đã được nói cho đến nay bằng một vài ví dụ.

Ví dụ 1. Độ tin cậy trên đường

Một đồ thị có trọng số đại diện cho một mạng máy tính được đưa ra, trong đó trọng số của mỗi cạnh $f(i, j)$ cho biết độ tin cậy của kết nối giữa các máy tính i và j (số từ 0 đến 1). Sau đó, độ tin cậy của đường dẫn trong đồ thị được xác định là tích của trọng lượng của các đường cạnh mà nó chứa. Bài toán là tìm cách đáng tin cậy

nhất để truyền thông tin giữa hai máy tính. Ở đây chương trình của Floyd được áp dụng như sau:

$$\varphi^{(k)}(i, j) = \max(\varphi^{(k-1)}(i, j), \varphi^{(k-1)}(i, k) \cdot \varphi^{(k-1)}(k, j))$$

Ma trận A[], thực hiện lược đồ, chứa 0 nếu không có cạnh giữa hai đỉnh. Vì giá (độ tin cậy) thu được mỗi lần bằng cách nhân trọng lượng của các cạnh của nó, nên hoạt động của nó \oplus là phép nhân. Vì đường đi với độ tin cậy cao nhất được tìm kiếm, F sẽ là *cực đại*.

Ví dụ 2. Đường có độ thâm lớn nhất

Trong đồ thị có trọng số đại diện cho hệ thống cấp nước, trọng lượng của mỗi cạnh $f(i, j)$ cho biết độ thâm của nó. Bài toán là tìm đường đi từ s đến t có độ thâm cực đại. Độ thâm của đường được định nghĩa là trọng lượng tối thiểu của cạnh nối hai đỉnh đường liên tiếp. Sơ đồ tổng quát của Floyd cho bài toán này sẽ là:

$$\varphi^{(k)}(i, j) = \max(\varphi^{(k-1)}(i, j), \min(\varphi^{(k-1)}(i, k), \varphi^{(k-1)}(k, j))),$$

Ma trận A[] chứa 0 nếu không có cạnh giữa hai đỉnh. Vì giá (độ thâm) của mỗi đường đi được xác định bởi cạnh có độ thâm thấp nhất nên hoạt động của nó là nhỏ nhất. Vì chúng ta đang tìm đường dẫn có thông lượng cao nhất, F sẽ là đường dẫn lớn nhất.

Ví dụ 3. Tìm t con đường nhỏ nhất đầu tiên

Việc tìm đường đi nhỏ nhất t đầu tiên giữa mỗi hai đỉnh trong đồ thị có thể được thực hiện lại bằng cách sử dụng lược đồ tổng quát của Floyd. Ứng dụng của nó trong trường hợp này không trực tiếp như vậy, nhưng lược đồ được giữ nguyên: hàm $\varphi^{(k)}(i, j)$ được định nghĩa không phải là một số mà là một vectơ của các số có ý nghĩa là độ dài của các đường đi nhỏ nhất. Tương tự, F và \oplus được xác định trên vectơ: cực tiểu của hai vectơ (x_1, x_2, \dots, x_t) và (y_1, y_2, \dots, y_t) là vectơ gồm nhiều nhất số t nhỏ giữa các số trong hai vectơ và tổng của hai vectơ: như một vectơ gồm t nhỏ nhất giữa các số $x_1 + y_1, x_1 + y_2, \dots, x_1 + y_t, x_2 + y_1, x_2 + y_2, \dots, x_2 + y_t, \dots, +x_t + y_1, x_t + y_2, \dots, x_t + y_t$.

Có một số ví dụ khác (có ứng dụng thực tế) trong đó lược đồ Floyd tổng quát được áp dụng thành công: và những ví dụ khác.

Nếu chúng ta tìm khoảng cách tối thiểu từ một đỉnh cố định s đến tất cả các đỉnh khác, thì có một thuật toán tốt hơn Floyd.

Bài tập

- ▷ 5.18. Viết chương trình tính toán độ tin cậy của đường (Ví dụ 1).
- ▷ 5.19. Viết chương trình tính toán độ thâm của đường (Ví dụ 2).
- ▷ 5.20. Viết chương trình để tìm p đường đi tối thiểu đầu tiên (Ví dụ 3).

- Thuật toán Dijkstra

Phương pháp hiệu quả nhất để tìm các đường đi nhỏ nhất từ một đỉnh cụ thể đến tất cả các đỉnh khác là thuật toán Dijkstra. Cho một đồ thị có hướng có trọng số $G(V, E)$ với n đỉnh được cho. Để áp dụng thuật toán, trọng số của các cạnh $f(i, j)$ phải là số dương (xem chú thích ở cuối đoạn). Hãy tìm đường đi nhỏ nhất từ một đỉnh cố định $s \in V$ đến tất cả các đỉnh khác của đồ thị. Với $\varphi(s, i)$, chúng ta sẽ biểu thị độ dài của đường đi nhỏ nhất từ s đến i . Nói chung, thuật toán Dijkstra dựa trên nguyên tắc sau: Để tìm $\varphi(s, i)$, chúng ta phải tìm giá trị nhỏ nhất trong số $\varphi(s, j) + f(j, i)$, với mỗi $j, j \neq i$. Do đó, mỗi đỉnh i của đồ thị được gán một giá trị thời gian $d[i]$, là giới hạn trên của $\varphi(s, i)$. Trong quá trình hoạt động của thuật toán, giá trị này giảm dần cho đến khi cuối cùng $d[i]$ trở thành chính xác bằng $\varphi(s, i)$:

Thuật toán Dijkstra

- 1) Khởi tạo mảng $d[]$ như sau:

$d[i] = A[s][i]$ cho mỗi $i \in V$ hàng xóm của s .

$d[i] = MAX_VALUE$, với mỗi đỉnh i , không kề với s .

Sau khi hoàn thành thuật toán $d[i] == MAX_VALUE$ t.s.t.k. không có đường đi giữa s và i .

- 2) Chúng ta giới thiệu một tập hợp T , ban đầu chứa tất cả các đỉnh của đồ thị, không có $s : T = V \setminus \{s\}$.

3) Trong khi T chứa ít nhất một đỉnh i mà $d[i] < MAX_VALUE$:

3.1) Ta chọn đỉnh $j \in T$ sao cho $d[j]$ là cực tiểu.

3.2) Chúng ta loại trừ j khỏi $T : T = T \setminus \{j\}$

3.3) Với mỗi $i \in T$ ta thực hiện $d[i] = \min(d[i], d[j] + A[j][i])$;

Để khôi phục các đỉnh liên quan đến đường dẫn nhỏ nhất (và không chỉ độ dài đường dẫn), chúng ta sẽ giới thiệu một mảng bổ sung pred[]. Do đó, ở vị trí thứ i của mảng pred[], đỉnh j được viết, mà $\varphi(s, j) + f(j, i)$ là cực tiểu:

```
if (d[i] > d[j] + A[j][i]) {
    d[i] = d[j] + A[j][i];
    pred[i] = j;
}
```

Trên mảng này, sau khi hoàn thành thuật toán, đường dẫn có thể được khôi phục, ví dụ bằng cách sử dụng hàm sau:

```
void printPath(unsigned s, unsigned j) {
    if (pred[j] != s)
        printPath(s, pred[j]);
    printf("%"d ", j); /* in được thực hiện sau khi quay trở lại từ đệ quy*/
}
```

Kỹ thuật được sử dụng trong chức năng trên để khôi phục và in đường đã xây dựng mà chúng ta đã biết từ 5.4.1 - ngầm định một ngăn xếp được sử dụng, bao gồm mỗi đỉnh kế tiếp và việc in được thực hiện sau khi kiểm tra đỉnh cuối cùng, trong quá trình đệ quy ngược lại.

Các đầu vào mẫu được sử dụng trong quá trình triển khai bên dưới được ghi lại dưới dạng hằng số ở đầu chương trình và tương ứng với cột trong Hình 5.11.

Chương trình 5.8. Thuật toán dijkstra (508dijkstra.c)

```
#include <stdio.h>
/* Số đỉnh tối đa trong đồ thị */
#define MAXN 150
#define MAX_VALUE 10000
#define NO_PARENT (unsigned)(-1)
/* Số đỉnh trong đồ thị */
const unsigned n = 10;
const unsigned s = 1;
/* Ma trận trọng số của đồ thị */
const unsigned A[MAXN][MAXN] = {
{ 0, 23, 0, 0, 0, 0, 0, 8, 0, 0 },
```

```

{23, 0, 0, 3, 0, 0, 34, 0, 0, 0 },
{ 0, 0, 0, 6, 0, 0, 0, 25, 0, 7 },
{ 0, 3, 6, 0, 0, 0, 0, 0, 0, 0 },
{ 0, 0, 0, 0, 0, 10, 0, 0, 0, 0 },
{ 0, 0, 0, 0, 10, 0, 0, 0, 0, 0 },
{ 0, 34, 0, 0, 0, 0, 0, 0, 0, 0 },
{ 8, 0, 25, 0, 0, 0, 0, 0, 0, 30 },
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
{ 0, 0, 7, 0, 0, 0, 0, 30, 0, 0 }
};

char T[MAXN];
unsigned d[MAXN];
int pred[MAXN];

/* Thuật toán Dijkstra - đường dẫn tối thiểu từ s đến các đỉnh khác */
void dijkstra (unsigned s)
{
    unsigned i;
    for (i = 0; i < n; i++) /*sự khởi tạo: d[i]=A[s][i], i trong V, i != s */
        if (0 == A[s][i]) {
            d[i] = MAX_VALUE;
            pred[i] = NO_PARENT;
        }
    else {
        d[i] = A[s][i];
        pred[i] = s;
    }
    for (i = 0; i < n; i++) T[i] = 1; /* T chứa tất cả các ngọn*/
    T[s] = 0;
    pred[s] = NO_PARENT; /* từ đồ thị, ngoại trừ s */
    while (1) { /* trong khi T chứa i: d[i] < MAX_VALUE */
        /*chọn đỉnh j từ T mà d[j] là cực tiểu*/
        unsigned j = NO_PARENT;
        unsigned di = MAX_VALUE;
        for (i = 0; i < n; i++)
            if (T[i] && d[i] < di) {
                di = d[i];
                j = i;
    }
}

```

```

        }
    if (NO_PARENT == j) break; /* d[i] = MAX_VALUE, với tất cả i:
        thoát*/
    T[j] = 0; /*Loại trừ j khỏi T */
    /*đối với mỗi i trong số T, thực hiện D[i] = min (d[i], d[j]+A[j][i
     ])
    */
    for (i = 0; i < n; i++)
        if (T[i] && A[j][i] != 0)
            if (d[i] > d[j] + A[j][i]) {
                d[i] = d[j] + A[j][i];
                pred[i] = j;
            }
    }
}

void printPath(unsigned s, unsigned j)
{ if (pred[j] != s) printPath(s, pred[j]);
    printf("%u ", j+1);
}

/* In các đường dẫn tối thiểu được tìm thấy*/
void printResult(unsigned s)
{ unsigned i;
    for (i = 0; i < n; i++) {
        if (i != s) {
            if (d[i] == MAX_VALUE)
                printf("Không có đường đi giữa các đỉnh %u và %u\n", s+1,
i+1);
            else {
                printf("Đường tối thiểu từ đỉnh %u đến %u: %u ", s+1, i+1, s
+1);
                printPath(s, i);
                printf(", chiều dài con đường: %u\n", d[i]);
            }
        }
    }
}

int main()
{ dijkstra (s-1); printResult(s-1);
}

```

}

Kết quả thực hiện chương trình:

Đường tối thiểu từ đỉnh 1 đến đỉnh 2: 1 2, chiều dài đường: 23

Đường tối thiểu từ đỉnh 1 đến 3: 1 2 4 3, chiều dài đường: 32

Đường tối thiểu từ đỉnh 1 đến 4: 1 2 4, chiều dài đường: 26

Không có đường đi giữa các đỉnh 1 và 5

Không có đường đi giữa các đỉnh 1 và 6

Đường tối thiểu từ đỉnh 1 đến 7: 1 2 7, chiều dài đường: 57

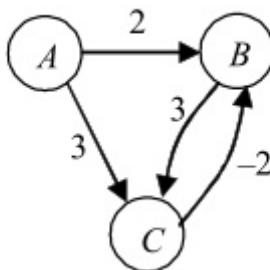
Đường tối thiểu từ đỉnh 1 đến 8: 1 8, chiều dài đường: 8

Không có đường đi giữa các đỉnh 1 và 9

Đường tối thiểu từ đỉnh 1 đến 10: 1 8 10, chiều dài đường: 38

Độ phức tạp của thuật toán trong cách triển khai trên là $\Theta(n^2)$. Với việc lựa chọn cẩn thận hơn các cấu trúc dữ liệu (ví dụ, nếu chúng ta sử dụng kim tự tháp - xem 3.1.9., [Cormen, Leiserson, Rivest-1997]) thì độ phức tạp có thể giảm xuống $\Theta(n \log_2 n)$.

Lưu ý: Thuật toán Dijkstra sẽ không hoạt động bình thường nếu có trọng số âm trong cột. Để minh họa điều này, chúng ta sẽ xem xét một ví dụ cụ thể (xem Hình 5.12).



Hình 5.12. Đồ thị có trọng số với cạnh âm.

Để tìm các đường đi nhỏ nhất A đến B và C, các bước sau sẽ được thực hiện:

- 1) Khởi tạo $d[A] = 0, d[B] = f(A, B) = 2, d[C] = f(A, C) = 3$.
- 2) Khởi tạo tập hợp $T = \{A, B, C\} \setminus \{A\} = \{B, C\}$.
- 3) Chọn một đỉnh $i \in T$ mà $d[i]$ là cực tiểu, đây là đỉnh B: Loại trừ B khỏi T. Với mỗi đỉnh từ T (và trong T chỉ có đỉnh C) được thực

hiện:

$$d[C] = \min(d[C], d[B] + f(B, C)) = \min(3, 2 + 3) = 3.$$

Trong bước tiếp theo, chỉ có C trong T, và sau khi tắt C, tập T vẫn trống và thuật toán kết thúc.

Vì vậy, chúng ta có $d[A] = 0, d[B] = 2, d[C] = 3$. Tuy nhiên, rõ ràng, đường đi ngắn nhất từ A đến B không phải là độ dài 2, mà là độ dài 1 (đây là đường đi ACB, vì $3 + (-2) = 1$).

Bài tập

- ▷ 5.21. So sánh thuật toán của Dijkstra với thuật toán của Ford-Bellman và Floyd.
- ▷ 5.22. Có thể khôi phục đường đi nhỏ nhất giữa một cặp đỉnh chỉ từ mảng $d[]$ mà không cần sử dụng một mảng bổ sung như $\text{pred}[]$ không?

- Lũy thừa ma trận của phần tử lân cận

Có một mối liên hệ thú vị giữa phép toán lũy thừa ma trận lên bậc k và số đường có độ dài k giữa hai đỉnh trong một đồ thị vô hướng. Cho một đồ thị vô hướng $G(V, E)$ với ma trận lân cận $A[][]$. Cho A' là ma trận sao cho $A'[i][j]$ chứa số đường đi có độ dài k giữa các đỉnh i và j . Khi đó trong ma trận $A'' = A'.A$ giá trị của $A''[i][j]$ sẽ chính xác là số đường có độ dài $k + 1$ giữa hai đỉnh. Sự thật của câu lệnh này dựa trên quy tắc nhân ma trận:

$$A''[i][j] = \sum_{t=1}^n A'[i][t].A[t][j]$$

Vì $A'[i][t]$ là số đường có độ dài k giữa i và t , nên số đường có độ dài $k + 1$ giữa i và j sẽ tăng lên $A'[i][t]$ với điều kiện là cạnh (t, j) là từ đồ thị (tương đương với $A[t][j] == 1$). Vì vậy, chúng ta đã đến với những điều sau

Định lý 5.2. Cho một đồ thị vô hướng $G(V, E)$ với một ma trận lân cận A . Nếu chúng ta nâng A lên lũy thừa của k , thì $C[i][j] = A^k[i][j]$ sẽ là số

đường có độ dài k , nằm giữa các đỉnh i và j . Nếu $C[i][j] = 0$, thì không có đường đi nào giữa i và j có độ dài k .

Định lý cũng có giá trị trong trường hợp đồ thị có định hướng (và thậm chí là đồ thị đa phương), nhưng chỉ khi các giá trị trong ma trận lân cận không bao gồm các giá trị -1 , biểu thị cạnh "quay lại".

Kết quả thu được không chỉ hữu ích trong việc giải các bài toán tìm số đường mà còn cả "đạo hàm" của chúng: chu trình, các dạng thành phần khác nhau và hơn thế nữa. Hơn nữa, do đồ thị "ma trận lân cận", nhiều bài toán cổ điển của đại số liên quan đến ma trận được chuyển thành các bài toán trên đồ thị, kể cả những bài toán vẫn chưa giải được.

Bài tập

► 5.23. Hãy viết chương trình thực hiện công việc dựa trên định lý trên.

- Thuật toán Warshal và ma trận khả năng tiếp cận

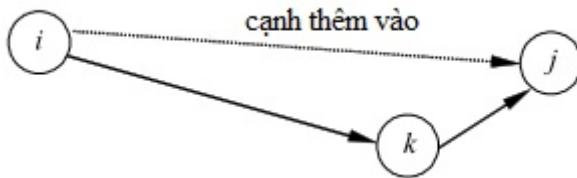
Bài toán: Giả sử một đồ thị có hướng $G(V, E)$ với một ma trận lân cận $A[[]]$. Chúng ta đang tìm một ma trận $A'[[]]$ sao cho:

- $A'[i][j] == 1$, nếu và chỉ khi có một đường đi (có độ dài bất kỳ) giữa các đỉnh i và j .
- $A'[i][j] == 0$, khi không có đường đi giữa hai đỉnh.

Thuật toán giải bài toán này được đặt tên theo Warshal và kết quả ứng dụng của nó - ma trận $A'[[]]$, được gọi là *ma trận khả năng phản ứng* của đồ thị. Đồ thị $G'(V, E')$ tương ứng với ma trận $A'[[]]$ (nếu chúng ta coi $A'[[]]$ là *ma trận lân cận*) được gọi là đồ thị chuyển tiếp đóng của G (nó sẽ được thảo luận lại trong 5.5.1.).

Thuật toán của Worschal là phức tạp $\Theta(n^3)$ và tiến hành như sau: với mỗi ba đỉnh $k, i, j \in V$, nếu $(i, k) \in E$ và $(k, j) \in E$, thì chúng ta thêm vào tập từ các cạnh của đồ thị và cạnh (i, j) (Hình 5.13).

Việc xem xét ba đỉnh được thực hiện bởi ba chu trình lồng nhau như sau:



Hình 5.13. Bước đóng chuyển tiếp.

Ba chu trình lồng nhau

```

for (k = 0; k < n; k++)
    for (i = 0; i < n; i++) {
        if (A[i][k])
            for (j = 0; j < n; j++)
                if (A[k][j]) A[i][j] = 1;
    }
}

```

Sau khi thực hiện đoạn chương trình được hiển thị, ma trận $A[][]$ sẽ được sửa đổi thành công thành ma trận có khả năng truy xuất yêu cầu $A'[][]$. (Tại sao?) Người đọc chú ý có lẽ đã nhận thấy sự giống nhau giữa thuật toán trên và Floyd. Đây không phải là ngẫu nhiên: Thuật toán Worshal là một trường hợp đặc biệt của Floyd.

Bài tập

- ▷ 5.24. Viết trong C/C++ thuật toán Worshal.
- ▷ 5.25. Chứng minh rằng thuật toán Worshal tìm đúng ma trận khả năng tiếp cận.

- đường đi dài nhất trong thị chu trình

Nhiều bài toán thực tế được giải quyết, rút gọn thành bài toán tìm đường đi dài nhất trong đồ thị mạch hở. Điều kiện của sự nhạy bén là điều cần thiết, vì cách tiếp cận duy nhất để giải quyết vấn đề trong trường hợp chung là thông qua sự kiệt sức hoàn toàn. Mặt khác, khi không có chu trình trong đồ thị, một số nguyên tắc tối ưu được đáp ứng (điều này cũng cần thiết cho tất cả các bài toán được giải bằng tối ưu hóa động - Chương 8) và bài toán có thể được giải

quyết hiệu quả hơn nhiều.

Chúng ta sẽ xem xét một phát biểu thực tế về bài toán tìm ra con đường tối đa.

Bài toán: Một nhóm lập trình viên phát triển một sản phẩm phần mềm bao gồm các bài toán riêng lẻ. Mỗi bài toán có một thời lượng cụ thể và kết nối hai giai đoạn phát triển sản phẩm: ban đầu và cuối cùng. Một bài toán không thể được bắt đầu nếu giai đoạn đầu tiên của nó chưa được hoàn thành. Để hoàn thành một giai đoạn, tất cả các bài toán cuối cùng của nó phải được hoàn thành. Để xác định thời gian tối thiểu (với điều kiện chúng ta có số lượng lập trình viên không giới hạn) đủ để hoàn thành toàn bộ dự án. Dự án được coi là hoàn thành khi tất cả các giai đoạn của nó đã được hoàn thành.

Nếu chúng ta sử dụng đồ thị hướng có trọng số trong mô hình bài toán, trong đó đỉnh là giai đoạn và cạnh là bài toán, thì thời gian tối thiểu cần thiết sẽ *bằng độ dài của số đỉnh lớn nhất* trong đồ thị. Trong tài liệu, con đường này còn được gọi là *con đường phê bình*.

Thuật toán dưới đây dựa trên kỹ thuật tối ưu hóa động (xem Chương 8).

Thuật toán:

Cho đồ thị có hướng chu trình $G(V, E)$ và $A[N][N]$ là ma trận các trọng số của G .

1) Chúng ta nhập một mảng $\text{maxDist}[]$, trong đó chúng ta sẽ giữ khoảng cách tối đa từ mỗi đỉnh trong cột, tức là $\text{maxDist}[i]$ sẽ bằng độ dài của đường đi lớn nhất bắt đầu với đỉnh i . Lúc đầu, chúng ta khởi tạo tất cả các phần tử của $\text{maxDist}[]$ bằng các số không. Trong mảng thứ hai $\text{savePath}[]$, chúng ta sẽ giữ các ý định là đường dẫn dài nhất, vì mảng được khởi tạo bằng -1 .

2) Xét một đỉnh $i \in V$ mà từ đó không có cạnh nào nhô ra. Một đỉnh như vậy nhất thiết phải tồn tại, bởi vì đồ thị là dòng chu trình (hãy để người đọc xem xét điều này đến từ đâu!). Chúng ta xóa đỉnh này khỏi đồ thị và gán cho từng đỉnh trước đó của nó k :

$$\text{maxDist}[k] = \max\{\text{maxDist}[k], \text{maxDist}[i] + A[k][i]\}$$

3) Thực hiện bước 2) cho đến khi không còn đỉnh nào trong đồ thị. Khi đó giá trị lớn nhất của $\text{maxDist}[i]$, với $i = 1, 2, \dots, n$ sẽ là độ

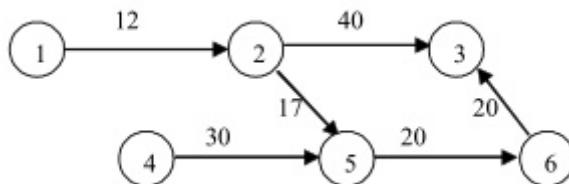
dài của đường dẫn tối đa được yêu cầu.

Thuật toán có độ phức tạp $\Theta(n + m)$, và khi biểu diễn đồ thị bằng ma trận lân cận - $\Theta(n^2)$. Chúng ta sẽ minh họa cách nó được áp dụng cho một đồ thị cụ thể. Hình 5.13. hiển thị thứ tự các đỉnh bị xóa khỏi đồ thị và các giá trị trong $\text{maxDist}[i]$ thay đổi như thế nào.

Thực hiện thuật toán:

Chúng ta sẽ thực hiện thuật toán được trình bày đê quy ở trên. Chúng ta sẽ sử dụng một sửa đổi của thu thập thông tin chuyên sâu. Mỗi lần chúng ta quay trở lại từ một lệnh gọi $DFS(i)$, chúng ta thấy mình đang ở trong một tiền nhiệm j ở đầu i . Ở đó, chúng ta sẽ giữ $\text{maxDist}[j] + A[j][i]$ tối đa cho mỗi thứ j kế thừa của i và cuối cùng, trước khi thoát khỏi hàm DFS , chúng ta sẽ gán nó cho $\text{maxDist}[i]$. Hàm $DFS(i)$ chỉ được thực thi khi $\text{maxDist}[i]$ chưa được tính, tức là. **if** $\text{maxDist}[i] == 0$ (khởi tạo ban đầu toàn bộ mảng bằng các số 0). Hàm DFS được khởi động tuần tự cho mỗi đỉnh của đồ thị.

Để lưu và in tất cả các đỉnh của đường tìm kiếm, không chỉ độ dài của nó, trong mảng $\text{savePath}[]$ ở vị trí thứ i , chúng ta sẽ viết đoạn kế tiếp này là j , mà $\text{maxDist}[i] + A[i][j]$ lớn nhất là thiết lập



Hình 5.14. Tìm đường quan trọng trong đồ thị.

1. Khởi tạo $\text{maxDist}[i] = 0$, cho mỗi i .
2. Xóa đỉnh 3 $\Rightarrow \text{maxDist}[6] = \max\{\text{maxDist}[6], \text{maxDist}[3] + 20\} = \max\{0, 20\} = 20$; Tương tự, $\text{maxDist}[2] = 40$
3. Xóa đỉnh 6 $\Rightarrow \text{maxDist}[5] = 40$;
4. Xóa đỉnh 5 $\Rightarrow \text{maxDist}[4] = 70, \text{maxDist}[2] = \max\{40, 40 + 17\} = 57$
5. Xóa đỉnh 4
6. Xóa đỉnh 2 $\Rightarrow \text{maxDist}[1] = 69$

7. Xóa đỉnh 1

Thời gian dài nhất: $70(maxDist[4])$

Sau đây là mã nguồn của chương trình. Số lượng đỉnh và ma trận trọng số $A[][]$ của đồ thị được đặt dưới dạng hằng số.

Chương trình 5.9. Đường dài nhất (509longpath.c)

```
#include <stdio.h>
/* Số đỉnh tối đa trong đồ thị */
#define MAXN 150
/* Số đỉnh trong đồ thị*/
const unsigned n = 6;
/*Ma trận trọng số của đồ thị*/
const unsigned A[MAXN][MAXN] = {
{ 0, 12, 0, 0, 0, 0 },
{ 0, 0, 40, 0, 17, 0 },
{ 0, 0, 0, 0, 0, 0 },
{ 0, 0, 0, 0, 30, 0 },
{ 0, 0, 0, 0, 0, 20 },
{ 0, 0, 20, 0, 0, 0 }
};

int savePath[MAXN], maxDist[MAXN];

void DFS(unsigned i)
{ int max, d;
unsigned j;

if (maxDist[i] > 0)
    return;
max = maxDist[i];
for (j = 0; j < n; j++)
    if (A[i][j]) {
        DFS(j);
        d = maxDist[j] + A[i][j];
        if (d > max) {
            max = d;
            savePath[i] = j;
        }
    }
maxDist[i] = max;
```

```

}

void solve(void)
{ unsigned i, maxi;
/*sự khởi tạo*/
for (i = 0; i < n; i++) {
    maxDist[i] = 0;
    savePath[i] = -1;
}

for (i = 0; i < n; i++)
    if (maxDist[i] == 0) DFS(i);

maxi = 0;
for (i = 0; i < n; i++)
    if (maxDist[i] > maxDist[maxi])
        maxi = i;

printf("Độ dài của đường dẫn tối hạn là% d \nĐường dẫn là:", maxDist[maxi]);
while (savePath[maxi] >= 0) {
    printf("%u ", maxi + 1);
    maxi = savePath[maxi];
}
printf("%d\n", maxi + 1);
}

int main() {
    solve();
    return 0;
}

```

Kết quả thực hiện chương trình:

Chiều dài của đường tối hạn là 70
Đường là: 4 5 6 3

Bài tập

- ▷ 5.26. Để chứng minh rằng trong một đồ thị mạch hở nhất thiết phải tồn tại:

- ít nhất một đỉnh không có đỉnh trước
- ít nhất một đỉnh không có người thừa kế

▷ 5.27. Để triển khai một biến thể của thuật toán, trong đó ở mỗi bước, một đỉnh không có đỉnh trước bị loại bỏ, thay vào đó là đỉnh không có đỉnh kế tiếp.

- Đường đơn dài nhất giữa hai đỉnh trong bất kỳ đồ thị nào

Chúng ta sẽ nhắc lại một lần nữa rằng việc tìm kiếm đường đi dài nhất trong đồ thị chu trình là một bài toán NP đầy đủ, mà chúng ta sẽ xem xét trong ???. Cạn kiệt hoàn toàn được áp dụng để giải quyết nó.

Bài tập

▷ 5.28. Đưa ra một ví dụ về một đồ thị có hướng và một cặp đỉnh của nó là i và j , trong đó thuật toán trước đó không tìm ra chính xác đường đi dài nhất giữa i và j .

▷ 5.29. Có những trường hợp nào khi thuật toán tìm đường đi dài nhất trong đồ thị chu trình sẽ hoạt động chính xác đối với một cặp đỉnh trong đồ thị tuần hoàn? Nếu có - khi nào? Nếu không - tại sao? Đưa ra các ví dụ có liên quan.

5.4.3. Chu trình

Tiếp tục logic của chủ đề từ 5.4.2 cho đường là việc tìm kiếm và nghiên cứu các chu trình của một đồ thị.

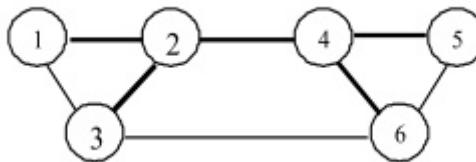
- tìm kiếm một tập hợp các chu trình cơ bản

Cho một đồ thị liên thông vô hướng $G(V, E)$ với n đỉnh và m cạnh. Cho $D(V, T)$ là một cây bao tùy ý của G . Thêm vào D một cạnh không thuộc T sẽ đóng một vòng lặp đơn giản. Chúng ta sẽ nói rằng chu trình này thuộc *tập các chu trình cơ bản* của đồ thị G đối với cây bao phủ D của nó.

Mỗi cây phủ D có đúng $q = n - 1$ cạnh. Các cạnh của G không thuộc D có tổng là $m - n + 1$. Vì khi thêm mỗi chúng vào T sẽ thu

được đúng một chu trình, nên số chu trình trong tập cơ bản là $m - n + 1$.

Xác định tập hợp các chu trình cơ bản, chúng ta xác định duy nhất cấu trúc chu trình của đồ thị, vì bất trình chu trình nào khác trong G đều có thể được biểu diễn bằng các chu trình "dán" từ một số tập cơ bản. Ví dụ, đối với các kết quả đọc trong Hình 5.15 đồ thị và cây che phủ đã chọn (các cạnh tham gia được hiển thị bằng màu tối) các chu trình đơn giản là $A = (1, 2, 3)$, $B = (4, 5, 6)$ và $C = (2, 3, 4, 6)$. Bất kỳ chu trình đơn giản nào khác đều có thể được xây dựng bằng cách nối (dán) một số chu trình A, B và C , có ít nhất một cạnh chung (chỉ riêng một đỉnh / đỉnh chung là không đủ!). Bằng cách nối hai chu trình, chúng ta có nghĩa là loại bỏ các cạnh chung của chúng (và tương ứng là các đỉnh cô lập sau khi loại bỏ các cạnh này) - vì vậy chúng ta có lại các chu trình đơn giản. Ví dụ, nối A và C , ta được chu trình $(1, 2, 4, 6, 3)$.



Hình 5.15. Tập hợp cơ bản của các chu trình trong đồ thị.

Chúng ta sẽ thực hiện việc tìm kiếm một tập hợp các chu trình cơ bản, một lần nữa bằng cách sử dụng sửa đổi chức năng thu thập thông tin độ sâu DFS . Trong bước đầu tiên, chúng ta sẽ tìm một cây bao phủ tùy ý của đồ thị, sau đó mỗi cạnh không tham gia vào cây bao phủ đã xây dựng sẽ đóng một chu trình và chúng ta sẽ tìm chu trình này với một đường đi ngang khác của đồ thị, được thực hiện bên dưới hàm $DFS2()$. Tổng độ phức tạp của thuật toán là $\Theta(m.(M + n))$.

Trong mã nguồn được đưa ra bên dưới, đồ thị được biểu diễn bằng ma trận lân cận $A[][]$, với $A[i][j]$ cho biết có cạnh giữa i và j hay không. Sau đó, trong quá trình thực thi chương trình, $A[i][j] == 2$, nếu cạnh tham gia vào cây bao phủ đã xây dựng.

Chương trình 5.10. Chu trình đơn (510allcyc.c)

```

#include <stdio.h>
/* Số đỉnh tối đa trong đồ thị */
#define MAXN 150
/* Số đỉnh trong đồ thị */
const unsigned n = 10;
/* * Đồ thị được biểu diễn bởi một ma trận lân cận: 0 - không có cạnh
   ;1 - có;
* Sau đó với 2 chúng ta sẽ đánh dấu các cạnh của cây của đồ thị.
*/
char A[MAXN][MAXN] = {
    { 0, 1, 1, 0, 0, 0 },
    { 1, 0, 1, 1, 0, 0 },
    { 1, 1, 0, 0, 0, 1 },
    { 0, 1, 0, 0, 1, 1 },
    { 0, 0, 0, 1, 0, 1 },
    { 0, 0, 1, 1, 1, 0 }
};

char used[MAXN];
unsigned cycle[MAXN], d;
/* Tìm bất kỳ cây phủ nào */
void DFS(unsigned v)
{
    unsigned i;
    used[v] = 1;
    for (i = 0; i < n; i++)
        if (!used[i] && A[v][i]) {
            A[v][i] = 2;
            A[i][v] = 2;
            DFS(i);
        }
}
/* In một chu trình tìm thấy*/
void printCycle(void)
{
    unsigned k;
    for (k = 0; k < d; k++)
        printf("%u ", cycle[k] + 1);
    printf("\n");
}
/* Tìm kiếm một chu trình theo cây phủ được tìm thấy */

```

```

void DFS2(unsigned v, unsigned u)
{ unsigned i;
  if (v == u) {
    printCycle();
    return;
  }
  used[v] = 1;
  for (i = 0; i < n; i++)
    if (!used[i] && 2==A[v][i]) {
      cycle[d++] = i;
      DFS2(i, u);
      d--;
    }
}

int main()
{
  unsigned i, j, k;
  DFS(0);
  printf("Các chu trình đơn trong đồ thị là: \n");
  for (i = 0; i < n - 1; i++)
    for (j = i + 1; j < n; j++)
      if (1 == A[i][j]) {
        for (k = 0; k < n; k++) used[k] = 0;
        d = 1;
        cycle[0] = i;
        DFS2(i, j);
      }
  return 0;
}

```

Kết quả thực hiện chương trình:

Các chu trình đơn trong đồ thị là:

1 2 3
2 3 6 4
5 4 6

Bài tập

- 5.30. Thuật toán được mô tả trên (có / không sửa đổi) có hoạt động đối với đồ thị đa phương không?

- 5.31. Thuật toán được mô tả trên (có / không sửa đổi) có hoạt động đối với đồ thị có định hướng không? Và cho một đồ thị nhiều định hướng?
- 5.32. Thuật toán được thảo luận ở trên có độ phức tạp $\Theta(m.(M + n))$, và việc triển khai, do sử dụng ma trận lân cận để biểu diễn đồ thị - độ phức tạp $\Theta(n^4)$. Có thể cải thiện kết quả này không? Chúng ta đã chỉ ra rằng số chu trình đơn có bậc là $\Theta(m + n)$, và độ dài của mỗi chu trình phụ thuộc vào chiều cao h của cây phủ đã xây dựng, tức là độ phức tạp tối thiểu để tìm và các vòng là $\Theta((m + n).h)$. Rõ ràng là trong trường hợp xấu nhất, chiều cao bằng với số đỉnh của đồ thị, tức là. độ phức tạp $\Theta((m + n).n)$ thu được.

Bạn có thể nghĩ ra một thuật toán để tìm các vòng lặp hoạt động với độ phức tạp tối thiểu được chỉ định $\Theta((m + n).h)$ không? Bạn có thể đề xuất chiến lược nào để xây dựng cây che phủ để nó có chiều cao tối thiểu? Có thể nói rằng với một chiến lược được lựa chọn tốt, h sẽ thuộc về $O(\log_2 n)$?

- chu trình tối thiểu qua đỉnh

Bài toán: Cho đồ thị có hướng trọng số $G(V, E)$ và đỉnh $i \in V$. Tìm một vòng lặp (không nhất thiết phải đơn) chứa đỉnh i và tổng trọng lượng của các cạnh tham gia là nhỏ nhất.

Lời giải: Với mỗi $j \in V$ ta tìm được đường đi nhỏ nhất $\varphi(i, j)$ từ i đến j và đường đi nhỏ nhất $\varphi(j, i)$ từ j đến i . Cho $S_k = \varphi(i, k) + \varphi(k, i), k \in V$. Khi đó S_k nhỏ nhất là độ dài của chu trình nhỏ nhất mà chúng ta đang tìm.

Điều quan trọng là chu trình được tìm thấy không nhất thiết phải đơn. Bài toán đặt ra một ràng buộc bổ sung cho chu trình để tránh các đỉnh lặp lại cũng khá thú vị. Nếu chúng ta xem xét một đồ thị không có trọng lượng của các cạnh, vẫn đề được giải quyết bằng cách thu thập thông tin theo chiều rộng: một sửa đổi nhỏ của thuật toán để tìm số đỉnh tối thiểu giữa hai đỉnh. Giải pháp trong trường hợp tổng quát (đối với đồ thị có trọng số) được thảo luận trong Bài toán 5.5.

Một bài toán thú vị khác về mặt thuật toán của các vòng lặp là

bài toán tìm một vòng lặp đơn tối thiểu chứa ít nhất k đỉnh (xem Bài toán 5.93).

Trường hợp "đặc biệt" của nó, tại $k = n$, được xem xét trong đoạn tiếp theo đây.

Bài tập

- ▷ 5.33. Triển khai thực hiện thuật toán trên được mô tả và tính độ phức tạp của nó.
- ▷ 5.34. Một đồ thị có hướng có trọng số và một đỉnh từ nó đã cho. Tìm một chu trình đơn chứa đỉnh mà tổng trọng lượng của các cạnh liên quan là nhỏ nhất.

5.4.4. Các chu trình Hamilton. Bài toán đường thương mại

Định lý 5.3. *Chu trình Hamilton (HC) trong đồ thị được gọi là chu trình chứa mỗi đỉnh của đếm chính xác một lần. Một đồ thị chứa một chu trình như vậy được gọi là một Hamilton.*

Mệnh đề 5.1. *Trong đồ thị Hamilton, chu trình Hamilton tương ứng có thể được xây dựng bắt đầu từ bất kỳ đỉnh ban đầu nào.*

Hai bài toán cổ điển là kiểm tra xem một biểu đồ có phải là Hamilton hay không và bài toán dành cho khách du lịch thương mại, đó là tìm một chu trình Hamilton có độ dài tối thiểu trong một biểu đồ có trọng số. Cả hai bài toán được định nghĩa là NP đầy đủ (xem ??), Có nghĩa là độ phức tạp là cấp số nhân trong trường hợp xấu nhất. Trong đoạn này chúng ta sẽ xem xét và giải quyết vấn đề thứ hai.

Thuật toán giải quyết công việc cho khách du lịch

Chúng ta sẽ giải quyết vấn đề bằng cách vét cạn hoàn toàn. Trong 5.4.1 chúng ta đã xem xét cách tìm tất cả các đường đi đơn giữa hai đỉnh. Chúng ta sẽ áp dụng cách tiếp cận tương tự, muôn các đỉnh bắt đầu và kết thúc trùng khớp. Chúng ta chọn bắt kỳ đỉnh i nào và bắt đầu từ nó, chúng ta sẽ xây dựng tất cả các tuyến có thể theo thứ tự. Chúng ta sẽ thay đổi độ dài $curSum$ của tuyến được xây dựng

cho đến nay trong phần thân của đệ quy. Khi chúng ta tìm thấy một chu trình Hamilton mới, chúng ta sẽ kiểm tra xem độ dài của nó có nhỏ hơn chu trình nhỏ nhất được tìm thấy cho đến nay hay không (chúng ta sẽ giữ nó trong biến $minSum$) và nếu nó nhỏ hơn - chúng ta sẽ lưu nó. Nếu trọng số của tất cả các cạnh của đồ thị là dương, chúng ta có thể làm cho thuật toán của mình hiệu quả hơn một chút bằng cách áp dụng "cắt đệ quy" (xem ??).

Kỹ thuật như sau: nếu tại bất kỳ bước nào chiều dài $curSum$ của con đường chúng ta đang xây dựng lớn hơn $minSum$, chúng ta sẽ làm gián đoạn việc xây dựng tuyến đường hiện tại. Điều này được thực hiện bởi vì ngay cả khi nó kéo dài sau chu trình Hamilton, thì độ dài của nó chắc chắn sẽ lớn hơn chiều dài nhỏ nhất. Tuy nhiên, để tận dụng tối ưu hóa này, đồ thị không được chứa các cạnh âm làm giảm $curSum$ sau đó! Để thấy điều này, chúng ta hãy nhìn vào ma trận lân cận mẫu sau:

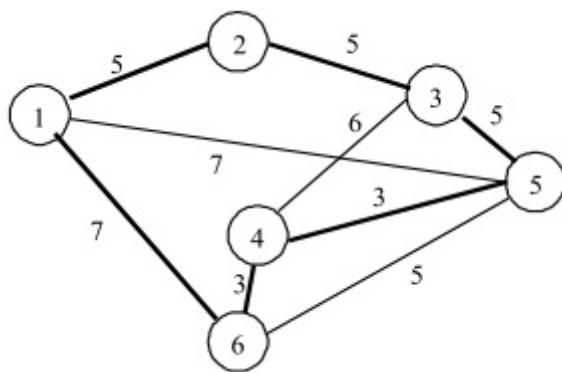
Ma trận lân cận mẫu

```
int A[n][n] =
{
    // A B C D
    { 0, 1, 2, 0 }, // A
    { -2, 0, 1, 0 }, // B
    { 0, 0, 0, 1 }, // C
    { 1, 2, 0, 0 } // D
};
```

Nếu ở một bước nào đó, chúng ta đã tìm thấy một chu trình Hamilton nhỏ nhất hiện tại (A, B, C, D, A) với độ dài $1 + 1 + 1 + 1 = 4$, thì khi chúng ta đạt đến đỉnh B trong chu trình Hamilton (A, C, D, B, A) tổng hiện tại sẽ là $2 + 1 + 2 = 5$. Và vì $5 > 4$, và 4 là độ dài tối thiểu cho đến nay, việc xem xét ứng cử viên mới, là tối thiểu, chấm dứt, bởi vì khi thêm cạnh, A) với trọng lượng - 2, độ dài của chu trình trở thành 3, và $3 < 4$.

Trong cách triển khai đề xuất, đồ thị được biểu diễn bằng ma trận trọng số $A[II]$, và dữ liệu đầu vào được ghi lại dưới dạng hàng số ở đầu chương trình. Đồ thị được sử dụng làm ví dụ về dữ liệu đầu vào trong quá trình thực hiện được thể hiện trong Hình 5.16.

Chương trình 5.11. Chu trình Hamilton nhỏ nhất (510tps.c)



Hình 5.16. Chu trình Hamilton nhỏ nhất

```

#include <stdio.h>
/* Số đỉnh có khả năng lớn nhất trong đồ thị*/
#define MAXN 150
#define MAX_VALUE 10000
/* Số đỉnh thật trong đồ thị */
const unsigned n = 6;
/*Ma trận trọng số của đồ thị*/
const int A[MAXN][MAXN] = {
    { 0, 5, 0, 0, 7, 7 },
    { 5, 0, 5, 0, 0, 0 },
    { 0, 5, 0, 6, 5, 0 },
    { 0, 0, 6, 0, 3, 3 },
    { 7, 0, 5, 3, 0, 5 },
    { 7, 0, 0, 3, 5, 0 }
};
char used[MAXN];
unsigned minCycle[MAXN], cycle[MAXN];
int curSum, minSum;

void printCycle(void)
{ unsigned i;
    printf("Chu trình Hamilton nhỏ nhất: ");
    for (i = 0; i < n - 1; i++) printf("%u", minCycle[i] + 1);
    printf(" 1, độ dài %d\n", minSum);
}

```

```

/* Tìm Chu trình Hamilton nhỏ nhất */
void hamilton(unsigned i, unsigned level)
{ unsigned k;
  if ((0 == i) && (level > 0)) {
    if (level == n) {
      minSum = curSum;
      for (k = 0; k < n; k++)
        minCycle[k] = cycle[k];
    }
    return;
  }
  if (used[i])
    return;
  used[i] = 1;
  for (k = 0; k < n; k++)
    if (A[i][k] && k != i) {
      cycle[level] = k;
      curSum += A[i][k];
      if (curSum < minSum) /*Gián đoạn vòng lặp*/
        hamilton(k, level + 1);
      curSum -= A[i][k];
    }
  used[i] = 0;
}

int main() {
  unsigned k;
  for (k = 0; k < n; k++) used[k] = 0;
  minSum = MAX_VALUE;
  curSum = 0;
  cycle[0] = 1;
  hamilton(0, 0);
  printCycle();
  return 0;
}

```

Kết quả thực hiện chương trình:

Chu trình Hamilton nhỏ nhất: 1 2 3 5 4 6 1, độ dài 28.

Thuật toán được trình bày và cách triển khai trên thực tế không thể áp dụng cho các đồ thị lớn: với khả năng tính toán hiện tại cho

đồ thị có hơn 50 đỉnh, có thể tìm thấy các ví dụ mà thuật toán sẽ hoạt động lâu đến mức không thể chấp nhận được. Do ứng dụng rộng rãi của nó, bài toán đã được nghiên cứu rất chi tiết trong các tài liệu [Reinelt-1994]. Có hàng tá (!) Trong số các thuật toán cho giải pháp của nó, một số trong số đó tìm ra nghiệm của độ phức tạp đa thức cho hầu hết các đồ thị (tất cả ngoại trừ logarit phụ thuộc vào n số đồ thị).

Trước khi chuyển sang loại chu trình thú vị tiếp theo trong đồ thị, chúng ta sẽ xem xét một câu hỏi khác liên quan đến các chu trình Hamilton và tìm kiếm của chúng. Đó là một câu hỏi làm thế nào để xác suất tồn tại của một chu trình Hamilton trong một đồ thị tùy ý thay đổi so với kết nối của đồ thị. Rõ ràng là một đồ thị đầy đủ (xem định nghĩa 5.8) Luôn luôn là Hamilton (bất kỳ sự sắp xếp nào của các đỉnh mà chúng ta lấy, chúng sẽ luôn tạo thành HC). Khi đồ thị gần như đầy đủ (chính xác hơn là gần như liên thông hoàn toàn), xác suất tồn tại HC là rất cao, số chu trình Hamilton cũng vậy. Con số này giảm mạnh khi số lượng cạnh giảm dần. Ở một cực khác, khi khả năng kết nối của đồ thị gần bằng 2, xác suất tồn tại của một chu trình Hamilton là nhỏ. Tại điểm tới hạn giữa hai điểm cực trị này, xác suất tồn tại của chu trình Hamilton phân bố đều giữa 0 và 1. Lý thuyết cho thấy rằng phân bố xác suất đồng đều này được thỏa mãn trong trường hợp chúng ta có kết nối trung bình của một đồ thị $\ln n + \ln \ln n$ [Christofides-1975].

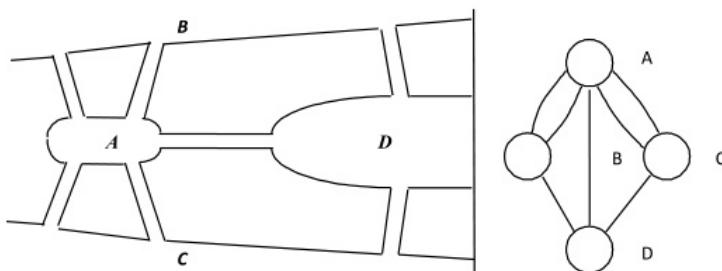
Bài tập

- ▷ 5.35. Hãy tìm một số lớp của đồ thị "đặc biệt" mà việc tìm kiếm chu trình Hamilton là "dễ dàng" (được thực hiện với độ phức tạp đa thức). Các ví dụ về vấn đề này được thảo luận trong ??.

5.4.5. Chu trình Euler

Sự khởi đầu của lý thuyết đồ thị được coi là sự xem xét của một loại chu trình đặc biệt, được đặt theo tên nhà nghiên cứu đầu tiên L. Euler. Thành phố Königsberg (nay là Kaliningrad) có bảy cây cầu bắc qua sông Pregel, được xây dựng như trong Hình 5.17.

Mọi người ở đó thường tự hỏi liệu có thể đi bộ qua thành phố để người ta có thể đi qua chỉ một lần trong những cây cầu và chuyền



Hình 5.17. 7 cầu ở Königsberg và đồ thị của chúng

tham quan sẽ kết thúc ở vị trí xuất phát hay không. Euler đã trình bày các cây cầu bằng đồ thị đa điểm (Hình 5.17) và chỉ ra rằng điều này là không thể.

Định nghĩa 5.16. Một đồ thị được kết nối được đưa ra. Chu trình mà mỗi cạnh tham gia đúng một lần được gọi là *chu trình Euler*. Một đa đồ thị được gọi là một đồ thị Euler nếu có một chu trình Euler trong đó.

Đường dẫn của Euler trong đồ thị cũng được xác định tương tự.

Định lý 5.4 (Euler). Một đồ thị đa hướng không định hướng được kết nối chứa một chu trình Euler, khi và chỉ khi tất cả các đỉnh của đồ thị có bậc chẵn. (được coi là định lý đầu tiên trong lý thuyết đồ thị được biểu diễn bởi Euler mà không cần chứng minh).

Bổ đề 5.1. Một đồ thị đa hướng không định hướng liên kết chứa một đường Euler khi và chỉ khi có đúng hai đỉnh có bậc lẻ.

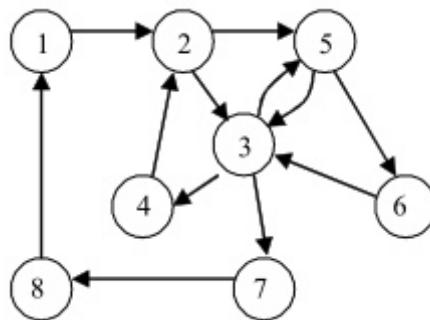
Định lý 5.5. Một đồ thị đa phương liên kết lồng chứa chu trình Euler khi và chỉ khi bậc độ của đầu vào $d^+(i)$ tại mỗi đỉnh i bằng bậc độ của đầu ra $d^-(i)$, tức là: $d^+(i) = d^-(i)$, với mỗi $i \in V$.

Bổ đề 5.2. Một đồ thị có định hướng được ghép nối lồng lỏi chứa đường đi Euler khi và chỉ khi có đúng hai đỉnh i và j sao cho $d^+(i) = d^-(i) + 1$, $d^+(j) = d^-(j) - 1$ và $d^+(k) = d^-(k)$, với mỗi $k \in V, k \in i, k \neq j$.

Mệnh đề 5.2. Trong đa đồ thị Euler, một chu trình Euler có thể được xây dựng bắt đầu từ bất kỳ đỉnh ban đầu nào.

Bổ đề 5.3. Nếu có một đường Euler trong một đồ thị đa phương được ghép nối lồng lẻo, thì đường ban đầu là đỉnh có bậc độ ở đầu ra lớn hơn bậc độ ở đầu vào.

Một ví dụ về đồ thị Euler có định hướng được thể hiện trong Hình ???. Một chu trình Euler có thể có trong nó là $(1, 2), (2, 3), (3, 4), (4, 2), (2, 5), (5, 3), (3, 5), (5, 6), (6, 3), (3, 7), (7, 8), (8, 1)$.



Hình 5.18. Chu trình Euler

Thuật toán tìm chu trình Euler

Chúng ta bắt đầu duyệt đồ thị từ đỉnh i bất kỳ. Chúng ta vô tình tìm thấy một cạnh (i, j) với nó và đánh dấu nó là đã thăm. Chúng ta tiếp tục với đỉnh j : đối với nó, chúng ta tìm thấy một cạnh không được chờ đợi (j, k) , đánh dấu nó là đã thăm và chuyển đến k . Tiếp tục theo cách này, đến một lúc nào đó chúng ta sẽ thấy mình ở đỉnh ban đầu i , tức là, chúng ta sẽ đóng chu trình (Tại sao?).

Nếu tất cả các cạnh của đồ thị đã được đánh dấu, thì chu trình này là Euler. Nếu có các cạnh không được thăm thì ta tìm được đỉnh x , thuộc chu trình mới tìm được và có ít nhất một cạnh không được thăm. Từ thực tế là mỗi đỉnh phải có mức độ chẵn, theo đó x là sự cố với một số chẵn (tức là ít nhất hai) các cạnh không được chọn. Từ x , chúng ta bắt đầu xây dựng một vòng lặp theo cách đã được mô tả, cho đến khi chúng ta quay trở lại nó. Chúng ta nhận được một chu trình thứ hai, mà chúng ta kết hợp với chu trình đầu tiên (điểm chung của chúng sẽ là đỉnh x). Do đó, sau một số bước hữu hạn, tất cả các sườn sẽ được đưa vào một chu trình chung - chu trình Euler mong muốn.

Nếu chúng ta tìm đường đi Euler, chúng ta có thể áp dụng cách sửa đổi đơn giản sau của thuật toán trên: chúng ta nối các đỉnh có bậc lẻ với một cạnh (từ Hệ quả 1 theo đó có đúng hai đỉnh như vậy) và tìm chu trình Euler bằng cách thuật toán được mô tả ở trên. Sau khi loại bỏ phần sườn đã thêm, chúng ta sẽ có được đường dẫn Euler mong muốn.

Cho đồ thị được biểu diễn bằng ma trận lân cận $A[][],$ Để thực hiện, chúng ta sẽ sử dụng hai ngăn xếp: $stack[],$ cho vòng lặp hiện được xây dựng và $cStack[]$ - để hợp nhất tất cả các vòng được xây dựng cho đến nay. Cả hai ngăn xếp ban đầu sẽ trống. Mã giả sau đây mô tả chi tiết hơn việc triển khai thuật toán:

```

<thêm đỉnh bất kỳ i vào stack>;
while (<stack khác trống>) {
    <Lấy một đỉnh i ở đỉnh stack không loại nó>;
    if (<i có đỉnh kề>) {
        <Lấy đỉnh kề bất kỳ j của i>;
        <đặt j stack>;
         $A[j][i] = 0;$   $A[i][j] = 0;$  /* Loại bỏ cạnh của đồ thị */
    } else {
        <Loại i khỏi stack>;
        <Đưa i vào cStack>;
    }
}

```

Triển khai đầy đủ sau đây. Đồ thị mẫu là của Hình 5.18.

Chương trình 5.12. Chu trình Euler (511euler.c)

```

#include <stdio.h>
/* Khả năng lớn nhất đỉnh của đồ thị */
#define MAXN 100
/* Số đỉnh thật của đồ thị*/
const unsigned n = 8;
/* Ma trận cạnh kề trong đồ thị */
char A[MAXN][MAXN] = {
    { 0, 1, 0, 0, 0, 0, 0, 0 },
    { 0, 0, 1, 0, 1, 0, 0, 0 },
    { 0, 0, 0, 1, 1, 0, 1, 0 },
    { 0, 1, 0, 0, 0, 0, 0, 0 },
    { 0, 0, 1, 0, 0, 1, 0, 0 },

```

```

{ 0, 0, 1, 0, 0, 0, 0, 0 },
{ 0, 0, 0, 0, 0, 0, 1 },
{ 1, 0, 0, 0, 0, 0, 0, 0 };
/*Kiểm tra xem có chu trình Euler không (theo định lý Euler)*/
char isEulerGraph(void)
{ unsigned i, j;
  for (i = 0; i < n; i++) {
    int din = 0, dout = 0;
    for (j = 0; j < n; j++) {
      if (A[i][j]) din++;
      if (A[j][i]) dout++;
    }
    if (din != dout) return 0;
  }
  return 1;
}

/*Tìm chu trình Euler*/
void findEuler(int i)
{ unsigned cStack[MAXN * MAXN], stack[MAXN * MAXN];
  unsigned k, j, cTop = 0, sTop = 1;
  stack[sTop] = i;
  while (sTop > 0) {
    i = stack[sTop];
    for (j = 0; j < n; j++)
      if (A[i][j]) {
        A[i][j] = 0; i = j;
        break;
      }
    if (j < n)
      stack[++sTop] = i;
    else
      cStack[++cTop] = stack[sTop--];
  }

  printf("Chu trình Euler là: ");
  for (k = cTop; k > 0; k--) {
    printf("%u ", cStack[k] + 1);
  }
  printf("\n");
}

```

```

    }

int main(){
    if (isEulerGraph()) findEuler(0);
    else
        printf("Đồ thị không phải là Euler!");
    return 0;
}

```

Kết quả thực hiện chương trình:

chu trình Euler là: 1 2 3 4 2 5 3 5 6 3 7 8 1

Bài tập

- ▷ 5.36. Hãy chứng minh định lý Euler và các hệ quả của nó.
- ▷ 5.37. So sánh bài toán tìm chu trình Hamilton với bài toán tìm chu trình Euler. Đâu là lý do cho việc đầu tiên khó hơn đáng kể so với lần thứ hai?

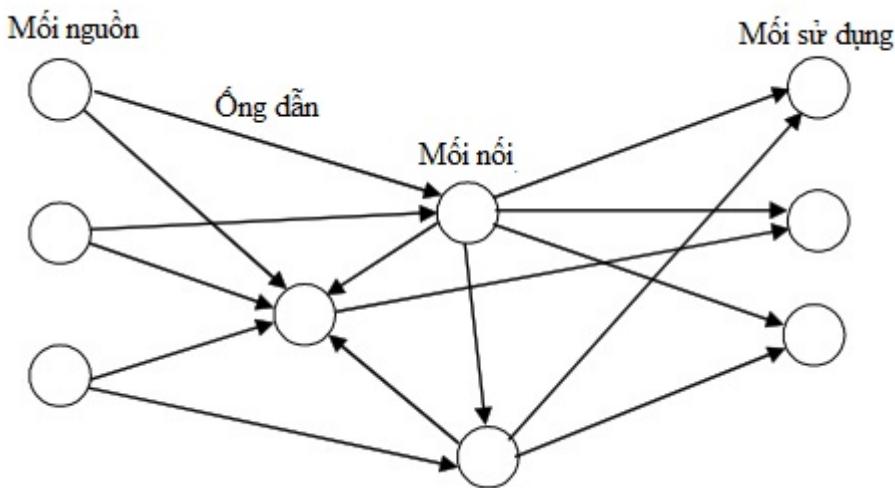
5.4.6. Luồng trong đồ thị

Giả sử một đồ thị có hướng $G(V, E)$ trong đó các đỉnh được chia thành ba tập hợp:

- nhiều nguồn nguyên liệu;
- nhiều người tiêu dùng;
- tập hợp các đỉnh trung gian - các kết nối mà qua đó vật liệu có thể được phân phối đến các kết nối hoặc người tiêu dùng khác.

Mỗi nguồn được so sánh với một con số - lượng nguyên liệu tối đa mà nó có thể phân phối và đối với mỗi người tiêu dùng được cung cấp số lượng nguyên liệu tối đa mà nó có thể nhận được. Các cạnh định hướng (i, j) của đồ thị có thể được hiểu là "đường ống" nối các đỉnh i và j , và đối với mỗi cạnh định hướng, chúng ta xác định:

- 1) Hàm c : $c(i, j)$ giới hạn lượng vật liệu có thể đi qua cạnh (i, j) , $c(i, j) \geq 0$, với mọi $(i, j) \in E$.
- 2) Hàm t : $t(i, j)$ thể hiện chi phí (chi phí) vận chuyển vật liệu qua sườn (i, j) .



Hình 5.19. Mô hình chuyển đổi

Bài toán chung của việc tìm ra bài toán dòng chảy mạng chi phí tối thiểu là tìm ra phương án vận chuyển nguyên liệu từ các nguồn đến người tiêu dùng sao cho tổng chi phí vận chuyển là nhỏ nhất. Năm 1960, Ford và Fulkerson đã đề xuất một thuật toán (gọi là kilter) để giải quyết vấn đề này. Bài toán có nhiều cách hiểu khác nhau và các trường hợp đặc biệt hữu ích, ví dụ:

Bài toán giá trị tối ưu

Cho trước là n máy, n bộ phận và ma trận vuông $A_{n \times n}$: a_{ij} cho biết chi phí chế tạo bộ phận i từ máy j . Tìm sơ đồ tạo các chi tiết (hoán vị p của các phần tử từ 1 đến n) sao cho tổng chi phí là nhỏ nhất (tức là tổng nhỏ nhất $\sum_{i \in V} a_{ip(i)}$.

Tác vụ này có thể được giảm xuống một tác vụ luồng tối thiểu như sau:

- Chúng ta coi máy móc là nguồn cung cấp một đơn vị vật chất.
- Chúng ta coi các chi tiết là người tiêu dùng yêu cầu một đơn vị vật liệu.
- Mỗi sườn có công suất đơn vị và giá bằng giá trị tương ứng trong bảng cho sẵn trong điều kiện.
- Không có đỉnh trung gian (kết nối).

Biến thể ngược lại của nguyên công cũng có thể xảy ra: bảng vuông cho thấy thu nhập từ việc sản xuất các chi tiết (tùy thuộc vào loại máy nào sẽ sản xuất nó) và chúng ta đang tìm kiếm lợi nhuận tối đa.

Ngoài việc giảm đến một luồng tối ưu, vấn đề thứ hai cũng có thể được giải quyết bằng cái gọi là thuật toán Hungarian [Shishkov-1995].

- Lưu lượng luồng cực đại

Một biến thể của bài toán trên là tìm dòng chảy tối đa. Các bài toán kiểu này được sử dụng trong các mạng thông tin liên lạc, trong việc lập kế hoạch cho các mạng chuyển vật chất (khí đốt, nước, nhiên liệu) và các mạng khác. Thuật toán giải nó đã được biết đến từ năm 1950 và một lần nữa thuộc về Ford và Fulkerson.

Chúng ta xem xét một đồ thị có trọng số $G(V, E)$, trong đó trên mỗi cạnh (i, j) một số không âm $c(i, j)$ được ánh xạ, nghĩa là dung lượng (thông lượng). Hai đỉnh đặc biệt được cố định trong đồ thị: nguồn s và người tiêu dùng t .

Định nghĩa 5.17. Luồng trong đồ thị được gọi là hàm $f : E \rightarrow Z$, phù hợp với số $f(i, j)$ trên mỗi cạnh (i, j) và có ba tính chất sau:

1) Lưu lượng không vượt quá khả năng của "đường ống", tức là. với mỗi cạnh $(i, j) \in E$, $f(i, j) \leq c(i, j)$ được thỏa mãn.

2) Nếu dòng chảy là âm, có nghĩa là có một "dòng chảy ngược". Nói chung, nếu luồng từ i đến j bằng $f(i, j)$, thì nhất thiết phải tuân theo luồng từ j đến i là $-f(i, j)$, với mỗi $(i, j) \in E$.

3) Độ lớn của dòng vào và dòng ra bằng nhau, tức là. với mỗi đỉnh $i \in V \setminus \{s, t\}$ thỏa mãn $\sum_{j:(i,j) \in E} f(i, j) = 0$.

Bài toán tìm giá trị của luồng cực đại từ s đến t , tức là một hàm f như vậy (thỏa mãn ba điều kiện trên) mà $\sum_{i:(i,t) \in E} f(i, t) = 0$ là cực đại.

Thuật toán Ford-Fulkerson

Ý tưởng chính của thuật toán là liên tục tăng luồng khi có thể:

1) Chúng ta bắt đầu với luồng không: $f(i, j) = 0$, với mỗi $(i, j) \in E$.

2) Chúng ta tìm thấy một *đường đi tăng* trong đồ thị. *Đường đi tăng dần* ($s = v_0, v_1, \dots, v_k = t$) là đường đi từ s đến t sao cho mọi hai đỉnh kề nhau v_i, v_{i+1} ($i = 0, 1, \dots, k - 1$) của đường đi được hoàn thành $c(v_i, v_{i+1}) > 0$.

3) Nếu một đường dẫn như vậy không tồn tại, thì theo đó chúng ta đã tìm thấy luồng cực đại trong đồ thị.

4) Nếu không, chúng ta tăng các luồng âm và dương lên số dương lớn nhất p , cho phép tìm được đường dẫn, tức là:

$$p = \min_{i=0, \dots, k-1} \{c(v_i, v_{i+1})\}$$

$$f(v_i, v_{i+1}) = f(v_i, v_{i+1}) + p, i = 0, 1, \dots, k - 1$$

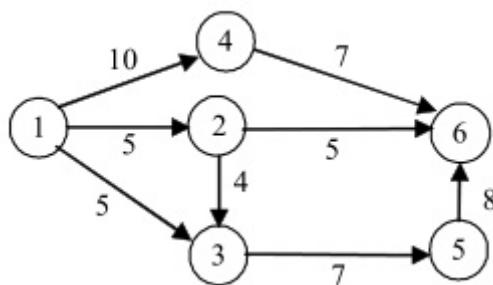
$$f(v_{i+1}, v_i) = f(v_{i+1}, v_i) - p, i = 0, 1, \dots, k - 1$$

Chúng ta sửa đổi trọng số (bảng thông) của các cạnh của biểu đồ như sau:

$$c(v_i, v_{i+1}) = c(v_i, v_{i+1}) - p, i = 0, 1, \dots, k - 1$$

$$c(v_{i+1}, v_i) = c(v_{i+1}, v_i) + p, i = 0, 1, \dots, k - 1$$

Chúng ta quay lại bước 2, nơi chúng ta tìm kiếm một con đường tăng mới, v.v.



Hình 5.20. Luồng cực đại trong đồ thị

Hãy nhìn vào đồ thị trong Hình 5.20., Và chọn nguồn $s = 1$ và người tiêu dùng $t = 6$. Thuật toán sẽ tìm ra 5 đường tăng liên tiếp:

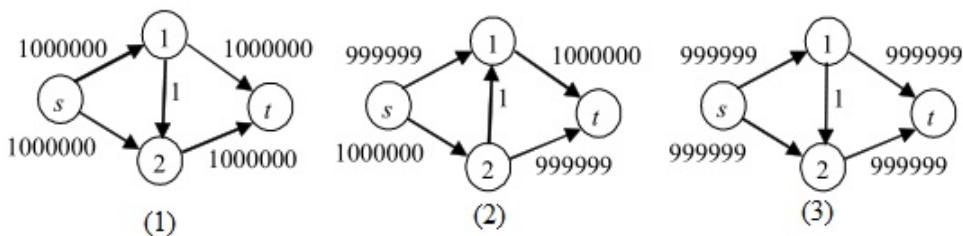
1. $(1, 2, 3, 5, 6)$: sườn tối thiểu của đường là $(2, 3)$ và trọng lượng của nó xác định mức tăng của tổng lưu lượng: trong trường

hợp là 4. Chúng ta sửa đổi trọng lượng của các sườn - ví dụ, từ đỉnh 2 đến đỉnh 3, chúng ta thu được dòng ngược chiều dài 4, dòng chảy này (xuất hiện ở bước 3) được sử dụng trong việc xây dựng đường tăng tiếp theo.

2. $(1, 2, 6)$: cạnh nhỏ nhất của con đường là $(1, 2)$ với trọng lượng 1 (vì chúng ta đã giảm trọng lượng của nó đi 4 ở bước 1).
3. $(1, 3, 2, 6)$: cạnh nhỏ nhất của đường là $(3, 2)$ với trọng số 4.
4. $(1, 3, 5, 6)$: cạnh nhỏ nhất của đường là $(1, 3)$ với trọng số 1.
5. $(1, 4, 6)$: cạnh nhỏ nhất của đường là $(4, 6)$ với trọng số 7.

Như vậy tổng kích thước của luồng lớn nhất là 17.

Có thể chứng minh rằng nếu không có đường đi tăng trong biểu đồ, thì dòng kết quả là cực đại [Christofides-1975]. Tuy nhiên, có thể xảy ra (nếu trong bước 2 của thuật toán bạn luôn chọn một con đường tăng dần tùy ý), độ phức tạp trong trường hợp xấu nhất sẽ đạt đến độ lớn của luồng tối đa, tức là sẽ phụ thuộc vào trọng số của các cạnh của đồ thị. Do đó, ngay cả đối với các đồ thị rất đơn giản (xem Hình 5.21), việc tìm ra lưu lượng cực đại có thể mất một thời gian dài ngoài dự kiến. Với ví dụ trong hình, độ phức tạp nhỏ nhất bằng độ lớn của luồng cực đại: Trong bước đầu tiên, chúng ta chọn một đường đi tăng dần $s - 1 - 2 - t$ và tăng luồng một. Chúng ta nhận được đồ thị (2), sau đó chúng ta đi theo đường tăng dần $s - 2 - 1 - t$ - chúng ta nhận được đồ thị (3), tương tự như đồ thị mà chúng ta đã bắt đầu, v.v. Giải pháp sẽ tìm ra 1.000.000 con đường gia tăng [Cormen, Leiserson, Rivest-1997].



Hình 5.21. Một ví dụ về sự kém hiệu quả trong việc chọn ngẫu nhiên một đường dẫn tăng.

Để giải quyết vấn đề thứ hai, chúng ta sẽ sử dụng định lý sau [Cormen, Leiserson, Rivest-1997]:

Định lý 5.6 (Luồng cực đại). Nếu đường được chọn ở bước 2) luôn là số lượng đỉnh tối thiểu liên quan, thì độ phức tạp của thuật toán trong trường hợp xấu nhất là $\Theta(n^5)$, tức là sẽ có nhiều nhất n^3 đường tăng, mỗi đường có độ phức tạp $\Theta(n^2)$.

Trong phương án sau đây, các đường đi tăng dần được định vị bởi truyền theo độ sâu, đồ thị được biểu diễn bằng ma trận trọng số A[][] và dòng chảy được giữ trong ma trận F[][].

Chương trình 5.13. Thuật toán Ford-Fulkerson (512fordfulk.c)

```
#include <stdio.h>
/* Số đỉnh cực đại có thể trong đồ thị*/
#define MAXN 100
#define MAX_VALUE 10000
const unsigned n = 10; /* Số đỉnh của đồ thị*/
const unsigned s = 1; /* Đỉnh nguồn*/
const unsigned t = 6; /* Đỉnh người sử dụng*/
/*Ma trận trọng số của đồ thị */
int A[MAXN][MAXN] = {
    { 0, 5, 5, 10, 0, 0 },
    { 0, 0, 4, 0, 0, 5 },
    { 0, 0, 0, 0, 7, 0 },
    { 0, 0, 0, 0, 0, 7 },
    { 0, 0, 0, 0, 0, 8 },
    { 0, 0, 0, 0, 0, 0 }
};
int F[MAXN][MAXN];
unsigned path[MAXN];
char used[MAXN], found;
```

```
void updateFlow(unsigned pl)
{ int incFlow = MAX_VALUE;
  unsigned i;
  printf("Đường tăng tìm được: ");
  for (i = 0; i < pl; i++) {
    unsigned p1 = path[i];
    unsigned p2 = path[i + 1];
    printf("%u ", p1+1);
    if (incFlow > A[p1][p2]) incFlow = A[p1][p2];
  }
```

```

printf("%u \n", path[pl]+1);
for (i = 0; i < pl; i++) {
    unsigned p1 = path[i];
    unsigned p2 = path[i + 1];
    F[p1][p2] += incFlow;
    F[p2][p1] -= incFlow;
    A[p1][p2] -= incFlow;
    A[p2][p1] += incFlow;
}
}

void DFS(unsigned i, unsigned level)
{ unsigned k;
if (found) return;
if (i == t-1) {
    found = 1;
    updateFlow(level - 1);
}
else
for (k = 0; k < n; k++)
if (!used[k] && A[i][k] > 0) {
    used[k] = 1;
    path[level] = k;
    DFS(k, level + 1);
    if (found) return;
}
}

int main()
{
unsigned i, j;
int flow;
/* 1) một luồng trống được khởi tạo*/
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++) F[i][j] = 0;
/* 2) tìm thấy một đường ngày càng tăng trong khi có thể*/
do {
    for (i = 0; i < n; i++) used[i] = 0;
    found = 0;
    used[s-1] = 1;
    path[0] = s-1;
}

```

```

DFS(s-1, 1);
} while (found);
/* In luồng ra */
printf("Lưu lượng tối đa qua đồ thị: \n");
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) printf("%4d", F[i][j]);
    printf("\n");
}
printf("\n");
flow = 0;
for (i = 0; i < n; i++) flow += F[i][t-1];
printf("Với độ lớn: %d\n", flow);
return 0;
}

```

Kết quả thực hiện chương trình:

Đã tìm thấy đường dẫn tăng: 1, 2, 3, 5, 6

Đã tìm thấy đường dẫn tăng: 1, 2, 6

Đã tìm thấy đường dẫn tăng: 1, 3, 2, 6

Đã tìm thấy đường dẫn tăng: 1, 3, 5, 6

Đã tìm thấy đường dẫn tăng: 1, 4, 6

Lưu lượng tối đa qua đồ thị:

0	5	5	7	0	0	0	0	0	0
-5	0	0	0	0	5	0	0	0	0
-5	0	0	0	5	0	0	0	0	0
-7	0	0	0	0	7	0	0	0	0
0	0	-5	0	0	5	0	0	0	0
0	-5	0	-7	-5	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

Với kích thước: 17

Chúng ta để người đọc sửa đổi chương trình trên để nó tìm ra các đường đi tăng dần bằng cách đi ngang theo chiều rộng, điều này sẽ đảm bảo độ dài đường đi tối thiểu, vì nó sẽ thỏa mãn điều kiện của định lý.

Ngoài ra, chúng ta sẽ nói rằng có các thuật toán để tìm luồng cực

đại trong đồ thị có độ phức tạp $\Theta(n^4)$ và thậm chí $\Theta(n^3)$ [Cormen, Leiserson, Rivest-1997].

Bài tập

► 5.38. Chứng minh rằng nếu không có đường đi tăng trong đồ thị, thì dòng kết quả là cực đại.

- Nhiều nguồn và người tiêu dùng

Nếu có nhiều hơn một nguồn và người tiêu dùng được đưa ra trong bài toán, chúng ta có thể dễ dàng giảm nó xuống nguồn vừa được xem xét. Gọi các nguồn là s_1, s_2, \dots, s_p và người tiêu dùng t_1, t_2, \dots, t_q . Chúng ta thêm một đỉnh mới s trong cột được gọi là siêu nguồn và một đỉnh mới t - siêu tích lũy. Ngoài hai mèo mới này, chúng ta thêm cạnh:

- (s, s_i) , sao cho $c(s, s_i) = +\infty$, với $i = 1, 2, \dots, p$.
- (t_j, t) , sao cho $c(t_j, t) = +\infty$, với $j = 1, 2, \dots, q$.

Sử dụng thuật toán Ford-Fulkerson từ đoạn trước, chúng ta tìm thấy luồng lớn nhất từ s đến t , sẽ bằng với tìm kiếm trong phiên bản gốc của bài toán. (Tại sao?)

Bài tập

► 5.39. Chứng minh rằng lưu lượng tối đa được tìm thấy bởi thuật toán Ford-Fulkerson trên đồ thị đã sửa đổi là lưu lượng tối đa cho đồ thị đầu ra, có một số nguồn và một số khách hàng.

- Công suất của các đỉnh

Có thể làm phức tạp thêm bài toán bằng cách đặt giới hạn cho lưu lượng đi qua mỗi đỉnh. Cho một hàm $v(i), v(i) \geq 0, i \in V$ sao cho:

$$\sum_{j \in V} f(i, j) \leq v(i), i \in V$$

Chúng ta đang tìm kiếm một dòng chảy tối đa từ nguồn s đến người tiêu dùng t . Ở đây, chúng ta cũng có thể giải quyết vấn đề

bằng cách giảm nó xuống bài toán tiêu chuẩn là tìm dòng chảy tối đa. Với mục đích này, chúng ta sẽ xây dựng một đồ thị mới $G'(V', E')$, trong đó:

1) Hai đỉnh của V' sẽ tương ứng với mỗi đỉnh $i \in V$: i_1 và i_2 , sẽ được nối với nhau bởi một cạnh (i_1, i_2) . Độ đi qua $c(i_1, i_2)$ của cạnh này sẽ bằng $v(i)$.

2) Mỗi cạnh (i, j) của G được chuyển đến G' là sườn (i_2, j_1) .

3) Trong đồ thị mới G' sẽ không có hàm số v giới hạn dòng chảy qua các đỉnh.

Như trong đoạn trước, luồng tối đa trong G' được tìm thấy bởi thuật toán Ford-Fulkerson sẽ là mức tối đa cho đồ thị G . (Tại sao?)

Các ứng dụng khác của bài toán luồng cực đại sẽ được thảo luận trong ?? và ??.

Bài tập

▷ 5.40. Chứng minh rằng lưu lượng tối đa được tìm thấy bởi thuật toán Ford-Fulkerson trên đồ thị đã sửa đổi là lưu lượng tối đa cho đồ thị đầu ra.

CHƯƠNG 5

LÝ THUYẾT ĐỒ THỊ

VÀ THUẬT TOÁN

5.5. Tính bắc cầu và cách xây dựng. Sắp xếp cấu trúc liên kết..	
411	
5.5.1. Đóng bắc cầu. Thuật toán Worschal	411
5.5.2. Định hướng bắc cầu	413
5.5.3. Giảm bắc cầu	418
5.5.4. Kiểm soát công ty	420
5.5.5. Bài tập	422
5.5.6. Sắp xếp theo cấu trúc liên kết	422
5.5.7. Sắp xếp tô pô đầy đủ	426
5.5.8. Bổ sung một đồ thị xoay chiều thành một đồ thị được liên kết yếu	430
5.5.9. Xây dựng đồ thị của các đỉnh đã cho	431
5.6. Khả năng tiếp cận và liên thông	432
5.6.1. Các thành phần liên thông	433
5.6.2. Các thành phần liên kết mạnh trong đồ thị định hướng	435
5.6.3. Điểm phân chia trong một đồ thị không định hướng	
439	
5.6.4. k –liên thông của đồ thị không định hướng	443
5.7. Các tập hợp con tối ưu và các tâm của đồ thị	444
5.7.1. Cây bao phủ tối thiểu	444
5.7.2. Tập đỉnh độc lập	455
5.7.3. Tập đỉnh trội	459
5.7.4. Tập cơ sở	463
5.7.5. Tâm, bán kính và đường kính	466
5.7.6. Kết hợp cặp. Kết hợp cặp tối đa	475
5.8. Tô màu và đồ thị phẳng	477
5.8.1. Tô màu đồ thị và sắc số	477
5.8.2. Đồ thị phẳng	479
5.9. Câu hỏi và bài tập	481

5.5. Tính bắc cầu và cách xây dựng. Sắp xếp cấu trúc liên kết

Ở phần đầu của chương, chúng ta nhận thấy rằng các tập hợp các đối tượng có quan hệ xác định giữa chúng có thể được biểu diễn bằng một đồ thị.

Một ví dụ đơn giản như vậy là tổ chức thứ bậc trong một tập đoàn. Mọi nhân viên trong đó đều là cấp dưới trực tiếp của người khác - những mối quan hệ như vậy có thể được biểu thị bằng một cái cây. Tuy nhiên, chúng thường phức tạp hơn. Ví dụ, ai đó có thể có nhiều hơn một "sếp" trực tiếp, hoặc có một chu trình. Trong những trường hợp như vậy cần sử dụng đồ thị có định hướng. Các bài toán có thể thực hiện là kiểm tra xem A có phải là cấp dưới của B (trực tiếp hay gián tiếp) hay không, hoặc liệt kê mọi người theo thứ bậc, tức là. đối với mỗi cặp (A, B) , nếu A là cấp dưới của B , thì A phải lùi xa trong danh sách hơn B .

Ví dụ thứ hai, hãy cho một bể cá và chúng ta muốn mua cá. Người ta biết loài nào có thể cùng tồn tại và loài nào không thể. Một bài toán khả thi là tìm ra số lượng tối đa các loài cá khác nhau có thể sống cùng nhau.

Các bài toán chúng ta sẽ xem xét trong đoạn này giải quyết các ví dụ được liệt kê, cũng như nhiều bài toán thực tế khác.

5.5.1. Đóng bắc cầu. Thuật toán Worschal

Trong ?? chúng ta đã sử dụng thuật toán Worschal khi chúng ta muốn tìm ra các cặp đỉnh trong đồ thị được nối với nhau bằng một đường dẫn. Từ ma trận lân cận A , chúng ta thu được ma trận khả năng truy xuất A' . Chúng ta đã nói rằng đồ thị G' với ma trận lân cận A' được gọi là *đóng bắc cầu* của G .

Trên đây chúng ta đã xem xét bài toán sắp xếp nhân viên trong tổng công ty theo hệ thống cấp bậc. Chúng ta hãy xây dựng một đồ thị có định hướng (và có thể theo chu trình) tương ứng $G(V, E)$

như sau: cạnh $(i, j) \in E$ tồn tại đỉnh i là trưởng trực tiếp của j . Nếu chúng ta tìm thấy bao đóng bắc cầu $G'(V', E')$ của G , thì $(i, j) \in E'$ tức là ta trưởng (trực tiếp hoặc gián tiếp) của j .

Có thể tìm thấy một số ứng dụng đóng chuyển tiếp khác. Như một bài tập đơn giản, chúng ta để người đọc xem xét cách nó có thể được áp dụng trong bài toán sau: Một tập hợp các sự kiện và một quan hệ được đã cho, cho biết sự kiện nào dẫn trực tiếp đến sự hiện thực của một sự kiện khác. Đối với một sự kiện t , chúng ta muốn tìm tập hợp các sự kiện S không thể xảy ra trước t . Ví dụ, để tìm vô số tất cả các sự kiện không thể xảy ra trước khi chúng ta đến tuổi trưởng thành.

Vai trò chính trong các bài toán được xem xét được đóng bởi tính chu trình trong đồ thị. Trong các ví dụ có thể được biểu diễn bằng đồ thị xoay chiều, việc tìm kiếm các quan hệ “gián tiếp” được giải quyết hiệu quả hơn nhiều (xem 5.5.6 - Sắp xếp theo cấu trúc liên kết). Chúng ta sẽ gọi lại thuật toán của Worschal từ ??:

```
for (k = 0; k < n; k++)
    for (i = 0; i < n; i++)
        if (A[i][k])
            for (j = 0; j < n; j++)
                if (A[k][j])
                    A[i][j] = 1;
```

Định lý 5.7 (Worschal). *Việc kiểm tra ba đỉnh (i, j, k) thông qua ba **for** các chu trình được lồng vào nhau theo cách được trình bày ở trên đảm bảo việc tìm ra chính xác sự đóng bắc cầu của đồ thị [Brassard, Bratley - 1996].*

Thoạt nhìn, định lý cuối cùng không có gì mới. Nó là một biến thể từ trên xuống (xem Chương 8) của lược đồ đệ quy trực quan tương ứng để tìm cách đóng bắc cầu:

$$A_k[i][j] = \begin{cases} A[i][j], & k = 0; \\ \min\{A_{k-1}[i][j], A_{k-1}[i][k] + A_{k-1}[k][j]\}, & k > 0. \end{cases}$$

Điều quan trọng là cần lưu ý thứ tự của các chu trình. Ví dụ: nếu thay vì phân đoạn ở trên, chúng ta thực thi:

```

for (i = 0; i < n; i++)
    for (k = 0; k < n; k++) if (A[i][k])
        .....

```

chúng ta sẽ nhận được một thuật toán mà trong trường hợp chung sẽ không tìm thấy điểm đóng bắc cầu chính xác của một đồ thị tùy ý. (Tại sao?)

Bài tập

- ▷ 5.41. Chứng minh định lý Worschal.
- ▷ 5.42. Chứng minh rằng các chu trình không thể đảo ngược. Nếu một đồ thị ví dụ trong đó chuyển vị dẫn đến bài toán.

5.5.2. Định hướng bắc cầu

Định nghĩa 5.18. Một đồ thị vô hướng $G(V, E)$ được cho. Định hướng các cạnh của nó được gọi là việc đã cho một thứ tự cho mỗi cạnh không định hướng (i, j) của E , tức là đồ thị thay đổi từ không định hướng sang có định hướng.

Định nghĩa 5.19. Cho một đồ thị có hướng $G(V, E)$, trong đó với mọi ba đỉnh $i, j, k \in V$ được thỏa mãn: Nếu $(i, k) \in E$ và $(k, j) \in E$, thì nó tuân theo cạnh $(i, j) \in E$. Khi đó G được gọi là bắc cầu.

Định nghĩa 5.20. Một đồ thị không có hướng $G(V, E)$ được gọi là có hướng chuyển tiếp nếu có thể định hướng các cạnh của nó theo cách mà đồ thị mới thu được có tính bắc cầu. Nếu không, đồ thị được gọi là không thể đọc chuyển tiếp.

Chúng ta sẽ xem xét bài toán định hướng bắc cầu của một đồ thị.

Thuật toán của Pnueli, Lempel và Ivena

Chúng ta sẽ giả định rằng đồ thị vô hướng đã cho được liên thông (nếu không, thì thuật toán được áp dụng tuần tự cho từng thành phần liên thông của nó).

Theo $i - j$, chúng ta có nghĩa là các đỉnh $i \in V$ và $j \in V$ được nối với nhau bởi một cạnh không có định hướng.

Theo $i \rightarrow j$, chúng ta có nghĩa là một cạnh được định hướng từ i đến j ; với $i \leftarrow j$, chúng ta có nghĩa là một cạnh được định hướng từ j đến i , và với $i \neq j$ là không có cạnh giữa i và j .

Chúng ta sẽ sử dụng hai quy tắc sau (đối với $i, j, k \in V$):

Quy tắc R1. Nếu thỏa mãn các điều kiện $i \rightarrow j, j - k, i \neq k$ thì ta định hướng cạnh $j - k : k \rightarrow j$.

Quy tắc R2. Nếu thỏa mãn các điều kiện $i \rightarrow j, i - k, j \neq k$ thì ta định hướng cạnh $i - k : i \rightarrow k$.

Thuật toán

- Chúng ta chọn một cạnh không định hướng ngẫu nhiên và cung cấp cho nó một số định hướng. Chúng ta đánh dấu cạnh được định hướng như đã xem xét để nếu chúng ta quay lại bước này lần thứ hai, chúng ta không chọn cùng một cạnh.

- Chúng ta áp dụng các quy tắc **R1** và **R2**, càng nhiều càng tốt, cho mỗi cạnh được định hướng:

Chúng ta hãy xem xét cạnh $i \rightarrow j$. Khi đó với mỗi đỉnh $k \in V$ kè với j ta chuyển sang trường hợp 2.1), và với mọi đỉnh k kè i - sang trường hợp 2.2).

2.1). Hãy nhìn vào cạnh (j, k)

a. (theo quy tắc R1) Nếu $i \neq k$ và $j - k$ được thỏa mãn, thì chúng ta định hướng (j, k) là $j \leftarrow k$

b. (mâu thuẫn với quy tắc R1) Nếu thỏa mãn $i \neq k$ và $j \rightarrow k$ thì đồ thị có tính bắc cầu không định hướng và kết thúc.

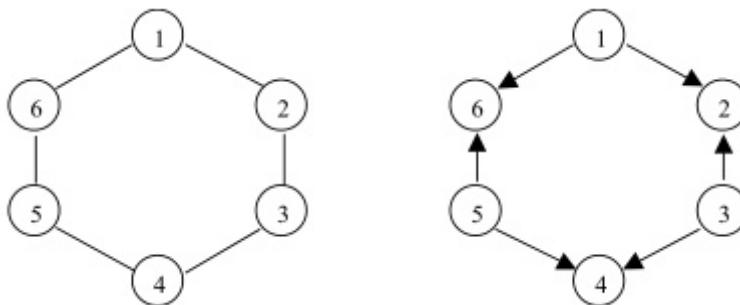
2.2). Xem xét cạnh (i, k)

a. (theo quy tắc R2) Nếu $j \neq k$ và $i - k$ được thỏa mãn, thì chúng ta định hướng (i, k) là $i \rightarrow k$.

b. (mâu thuẫn với quy tắc R2) Nếu $j \neq k$ và $i \leftarrow k$ được thỏa mãn thì đồ thị là không thể chuyển đổi được và chúng ta kết thúc.

- Kiểm tra xem tất cả các cạnh đều được định hướng. Nếu đúng như vậy, thì chúng ta đã có được hướng bắc cầu của đồ thị và chúng ta đã hoàn thành. Ngược lại, nếu không phải tất cả các cạnh đều được định hướng, chúng ta xóa các cạnh được định hướng khỏi đồ thị và quay lại bước 1).

Một ví dụ về định hướng bắc cầu được thảo luận trong [Hình 5.22](#).



Hình 5.22. Định hướng bắc cầu

Trong cách triển khai bên dưới, đồ thị được biểu diễn bằng ma trận lân cận $A[][]$. Nếu cạnh không định hướng (i, j) từ đồ thị thì $A[i][j] == 1$ và $A[j][i] == 1$. Ngược lại, giá trị của các phần tử này bằng 0. Khi chúng ta định hướng cạnh $i - j$ là $i \rightarrow j$, chúng ta gán $A[i][j] = 2$ và $A[j][i] = -2$ in ma trận. Vì ở bước 3) tất cả các cạnh định hướng đều bị xóa khỏi thuật toán, chúng ta phải lưu ý lưu chúng theo một cách nào đó, để sau khi hoàn thành thuật toán, có thể in được đồ thị có hướng chuyển tiếp. Với mục đích này, chúng ta sẽ sử dụng thêm hai giá trị: đối với tất cả các phần tử $A[i][j] == 2$ (và tương ứng là $A[j][i] == -2$), chúng ta sẽ gán $A[i][j] = -3$ và $A[j][i] = -4$. Do đó, việc kiểm tra xem cạnh (i, j) có phải từ đồ thị hay không sẽ là nếu ($A[i][j] > 0$).

Sau đây là mã nguồn của chương trình:

Chương trình 5.14. Định hướng bắc cầu (513trans-or.c)

```
#include <stdio.h>
/* Số đỉnh cực đại có thể của đồ thị */
#define MAXN 150
/* Số đỉnh đồ thị cho */
const unsigned n = 6;
/* Ma trận cạnh ef của đồ thị*/
int A[MAXN][MAXN] = {
    { 0, 1, 0, 0, 0, 1 },
    { 1, 0, 1, 0, 0, 0 },
    { 0, 1, 0, 1, 0, 0 },
    { 0, 0, 1, 0, 1, 0 },
    { 0, 0, 0, 1, 0, 1 },
```

```

{ 1, 0, 0, 0, 1, 0 }
};

/* // Ví dụ cho đồ thị không định hướng
const unsigned n = 5;
int A[MAXN][MAXN] = {
{ 0, 1, 0, 0, 1},
{ 1, 0, 1, 0, 0},
{ 0, 1, 0, 1, 0},
{ 0, 0, 1, 0, 1},
{ 1, 0, 0, 1, 0}};
*/



char trOrient(void)
{ /* tìm số cạnh trong đồ thi*/
    unsigned i, j, k, r, tr = 0;
    char flag;
    for (i = 0; i < n - 1; i++)
        for (j = i + 1; j < n; j++)
            if (A[i][j]) tr++;
    r = 0;
    do {
        for (i = 0; i < n; i++) { /* bước 1 - định hướng c (i,j) */
            for (j = 0; j < n; j++)
                if (1 == A[i][j]) {
                    A[i][j] = 2;
                    A[j][i] = -2;
                    break;
                }
            if (j < n) break;
        }
        /* Áp dụng quy tắc 1) và 2), đến khi nào có thể*/
        do {
            flag = 0;
            for (i = 0; i < n; i++) {
                for (j = 0; j < n; j++) {
                    if (2 == A[i][j]) {
                        for (k = 0; k < n; k++) {
                            if (i != k && j != k) {

```

```

if (0 == A[i][k] || A[i][k] < -2) { /*trường hợp 2.1*/
    /* a) -> đồ thị không bắc cầu định hướng*/
    if (2 == A[j][k]) return 1;
    /* b) -> định hướng cạnh (j,k) */
    if (1 == A[j][k])
        { A[k][j] = 2; A[j][k] = -2; flag = 1; }
    }
    if (0 == A[j][k] || A[j][k] < -2) { /*trường hợp 2.2*/
        /* a) -> đồ thị không bắc cầu định hướng*/
        if (2 == A[k][i]) return 1;
        /* b) -> cạnh định hướng (i,k) */
        if (1 == A[i][k]) {
            A[i][k] = 2;
            A[k][i] = -2;
            flag = 1;
        }
    }
}
}
}
}

} while (flag);

/*bước 3-loại các cạnh định hướng trong đồ thị*/
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        if (2 == A[i][j]) {
            A[i][j] = -3; A[j][i] = -4; r++;
        }
    } while (r < tr);
/* lặp lại cho đến khi tất cả cạnh của đồ thị là định hướng*/
return 0;
}

void printGraph(void)
{ unsigned i, j;
    printf("Định hướng bắc cầu là: \n");
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++)

```

```

if (-3 == A[i][j]) printf(" 1");
else (-4 == A[i][j]) ? printf(" -1") : printf(" 0");
printf("\n");
}
}

int main() {
if (trOrient())
printf("Đồ thị không định hướng bắc cầu! \n");
else
printGraph();
return 0;
}

```

Kết quả thực hiện chương trình:

Định hướng bắc cầu là:

```

0 1 0 0 0 1
-1 0 -1 0 0 0
0 1 0 1 0 0
0 0 -1 0 -1 0
0 0 0 1 0 1
-1 0 0 0 -1 0

```

Bài tập

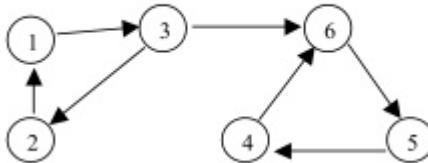
- ▷ 5.43. Hãy chứng minh rằng thuật toán của Pnueli, Lempel và Ivena hoạt động chính xác.

5.5.3. Giảm bắc cầu

Định nghĩa 5.21. Một đồ thị định hướng $G(V, E)$ với ma trận khả năng truy cập W được đã cho. Một đồ thị $G'(V, E')$, $E' \subseteq E$, với số cạnh tối thiểu mà ma trận tiếp cận của nó lại là W , được gọi là giảm bắc cầu của G .

Ví dụ, một trong những thể hiện trong Hình 5.23 (bên phải) đồ thị là sự giảm bắc cầu của bất kỳ đồ thị định hướng nào có chứa đồ thị đã cho và có ma trận khả năng truy cập được hiển thị ở bên trái.

	1	2	3	4	5	6
1	1	1	1	1	1	1
2	1	1	1	1	1	1
3	1	1	1	1	1	1
4	0	0	0	1	1	1
5	0	0	0	1	1	1
6	0	0	0	1	1	1



Hình 5.23. Giảm bắc cầu

Chúng ta sẽ xem xét bài toán tìm một giảm bắc cầu của một đồ thị có định hướng $G(V, E)$ cho trước với ma trận khả năng phản ứng W .

Thuật toán nghịch đảo của Worschal

Ta khởi tạo ma trận $A[][]$ với các giá trị như đã cho trong ma trận $W[][]$. Chúng ta loại bỏ liên tiếp các cạnh khỏi $A[][]$ cho đến khi chúng ta thu được độ giảm bắc cầu cần thiết (ở cuối thuật toán $A[][]$ sẽ là ma trận lân cận cho đồ thị cần thiết với số cạnh tối thiểu).

Với mỗi $k = 1, 2, \dots, n$ ta tìm được hai đỉnh i và j sao cho:

- có một cạnh (i, k) , tức là $A[i][k] == 1$.
- có một đường đi từ k đến j , tức là $W[k][j] == 1$.
- Nếu hai điểm trên gặp nhau và tồn tại cạnh (i, j) ($A[i][j] == 1$) thì nó bị loại bỏ vì nó thừa.

Thuật toán được thực hiện với ba chu trình lồng nhau và như với thuật toán Worschal, chu trình tính bằng k phải là chu trình bên ngoài nhất:

```

for (k = 0; k < n; k++)
    for (i = 0; i < n; i++)
        if (A[i][k])
            for (j = 0; j < n; j++)
                if (A[i][j] && W[k][j]) A[i][j] = 0;
  
```

Bài tập

- 5.44. Điều gì sẽ xảy ra nếu điều kiện $E' \subseteq E$ bị loại bỏ khỏi định nghĩa về giảm bắc cầu, tức là. Tìm một đồ thị có số cạnh nhỏ nhất (không nhất thiết phải là tập con của các cạnh đã cho) trên ma trận

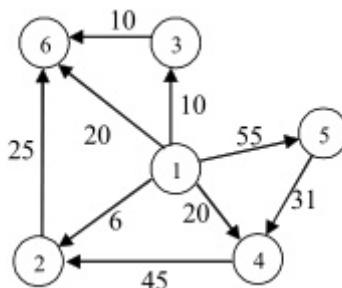
tiếp cận đã cho? Liệu thuật toán nghịch đảo của Worschal có hoạt động chính xác trong trường hợp này không?

5.5.4. Kiểm soát công ty

Bài toán: Một đồ thị có hướng trọng số $G(V, E)$ được cho với trọng số của các cạnh của số tự nhiên từ 0 đến 100. Ta nói rằng đỉnh i điều khiển j nếu tồn tại một cạnh $(i, j), f(i, j) > 50$ hoặc có nhiều đỉnh v_1, v_2, \dots, v_k do i điều khiển và có các cạnh $(v_1, j), (v_2, j), \dots, (v_k, j)$ và $f(v_1, j) + f(v_2, j) + \dots + f(v_k, j) > 50$. Bài toán là tìm tất cả các đỉnh do m điều khiển trên một đỉnh m cho trước.

Bài toán mô tả một tình huống thực tế trong đó các đỉnh của đồ thị đại diện cho các công ty và mỗi cạnh (i, j) cho thấy phần trăm cổ phần mà công ty i sở hữu từ công ty j .

Dữ liệu đầu vào mẫu cho bài toán được thể hiện trong Hình 5.5.4 trong Hình, Công ty 1 điều khiển 2, 4 và 5.



Hình 5.24. Đồ thị có trọng số định hướng.

Bài toán có thể được giải quyết bằng cách điều chỉnh thuật toán Worshal để đóng bắc cầu. Chúng ta sẽ giới thiệu một mảng $control[]$, trong đó chúng ta sẽ giữ tỷ lệ sở hữu của công ty i với các công ty khác. Lúc đầu, chúng ta khởi tạo mảng với trọng số của các cạnh từ i đến các đỉnh khác. Mỗi khi một công ty mới do ta kiểm soát xuất hiện, chúng ta sẽ thêm phần trăm mà công ty này kiểm soát từ những công ty khác vào mảng $control[]$. Một công ty mới do i kiểm soát xuất hiện khi tỷ lệ phần trăm ở vị trí $control[]$ trở nên trên 50. Độ phức tạp của thuật toán được mô tả là $\Theta(n^3)$.

Chương trình 5.15. Kiểm soát công ty (514company.c)

```

#include <stdio.h>
/* Số lượng công ty lớn nhất có thể (các đỉnh trong đồ thị) */
#define MAXN 150
/* Số ương công ty (số đỉnh trong đồ thị) */
const unsigned n = 6;
const unsigned m = 1; /* Tìm những công ty nào kiểm soát 1 công ty */
/*
/* Ma trận kề của đồ thị*/
const unsigned A[MAXN][MAXN] = {
    { 0, 6, 10, 20, 55, 20 },
    { 0, 0, 0, 0, 0, 25 },
    { 0, 0, 0, 0, 0, 10 },
    { 0, 45, 0, 0, 0, 0 },
    { 0, 0, 0, 31, 0, 0 },
    { 0, 0, 0, 0, 0, 0 }
};

unsigned control[MAXN];
char used[MAXN];

void addControls(void)
{ unsigned i, j;
  for (i = 0; i < n; i++)
    /*công ty i kiểm soát, ta thêm tác động của nó vào những người c
       ủa m */
    if (control[i] > 50 && !used[i]) {
      for (j = 0; j < n; j++)
        control[j] += A[i][j];
      used[i] = 1;
    }
}

void solve(void)
{ unsigned i;
  for (i = 0; i < n; i++) {
    control[i] = 0;
    used[i] = 0;
  }
  for (i = 0; i < n; i++) control[i] = A[m-1][i];
}

```

```

for (i = 0; i < n; i++) addControls();
}

void printResult(void) {
    unsigned i;
    printf("Công ty %u kiểm soát những công ty sau đây: \n", m);
    for (i = 0; i < n; i++)
        if (control[i] > 50)
            printf("%u: %3u% \n", i, control[i]);
    printf("\n");
}

int main() {
    solve();
    printResult();
}

```

Kết quả thực hiện chương trình:

Công ty 1 kiểm soát các công ty sau:

1: 53%
3: 51%
4: 55%

5.5.5. Bài tập

▷ 5.45. Có thể giải bài toán với độ phức tạp nhỏ hơn $\Theta(n^3)$ không?

5.5.6. Sắp xếp theo cấu trúc liên kết

Định nghĩa 5.22. *Sắp xếp tópô* của một đồ thị xoay chiều có định hướng $G(V, E)$ được gọi là danh sách có thứ tự tuyến tính Z gồm các đỉnh của nó, mà nó được thỏa mãn: Với mỗi hai đỉnh $i, j \in V$, nếu có một đường đi từ i đến j , thì đỉnh i phải đứng trước j trong Z .

Trong thực tế, danh sách Z hiếm khi có thể được xây dựng theo một cách duy nhất.

Định nghĩa 5.23. Khi có nhiều hơn một bậc của đỉnh trong Z , tập hợp tất cả các danh sách Z có thể có được gọi là *sắp xếp tópô* đầy đủ.

Mệnh đề 5.1. Một đồ thị mạch hở có định hướng $G(V, E)$ được cho. Chỉ có một danh sách Z , là cách sắp xếp theo cầu trúc liên kết của G , tức là G được liên thông yếu.

Ví dụ 1: Có một hệ thống cấp bậc nghiêm ngặt trong một tu viện ở Tây Tạng: mỗi tu sĩ có một hoặc nhiều cấp trên trực tiếp (người chịu trách nhiệm về những vi phạm của mình), và những người không có ông chủ được coi là cấp trên của họ. Các nghi lễ hiến tế được thực hiện hàng năm, và một số nhà sư phải được chọn để ném xuống "vực thẳm của vô cùng." Các nhà sư luôn được chọn để hiến tế trong số những người đứng thấp nhất trong hệ thống cấp bậc của tu viện. Tuy nhiên, thống đốc của tu viện đã không tốt với lý thuyết về số lượng và đã tìm thấy một lập trình viên để xác định những nhà sư q đó nên được hy sinh. Rõ ràng, bài toán là tìm cách sắp xếp tôpô của đồ thị và hy sinh q tu sĩ cuối cùng từ danh sách kết quả Z .

Ví dụ 2: Một ví dụ khác là sách giáo khoa toán. Các định lý, định nghĩa, tiên đề khác được sử dụng trong phần chứng minh của mỗi định lý. Sách giáo khoa nên được viết theo một cách nhất quán - "được sắp xếp theo cấu trúc liên kết" và mỗi định lý sẽ chỉ trích dẫn những điều đã được định nghĩa/chứng minh trước đó.

Vì không cho phép tính chu trình trong các đồ thị được coi là có định hướng, nên luôn có ít nhất một đỉnh trong chúng không có giá trị tiền nhiệm. (Tại sao?) Một đỉnh như vậy (lưu ý rằng có thể có nhiều hơn một) sẽ chiếm một vị trí hàng đầu trong sắc lệnh. Theo kết quả của kết luận vừa được đã cho, chúng ta nhận được

Thuật toán 1 để sắp xếp topo

1) Khởi tạo Z dưới dạng danh sách trống.

2) Chọn đỉnh i không có đỉnh trước và thêm nó vào cuối danh sách Z . Loại trừ i khỏi đồ thị, cũng như tất cả các đường liên quan đến nó.

3) Lặp lại bước 2) cho đến khi không còn đỉnh.

Ghi chú:

- Nếu ở bước 2) có nhiều hơn một đỉnh không có tiền nhiệm thì ta chọn tùy ý. (Tại sao?)

- Nếu ở bước 2) không có đỉnh nào không có đỉnh trước và có

nhiều đỉnh hơn trong đồ thị, thì đồ thị đó là tuần hoàn, điều này mâu thuẫn với điều kiện và do đó không có sắp xếp tópô. Điều ngược lại cũng đúng: nếu đồ thị là tuần hoàn, thì tại một số bước của thuật toán, chắc chắn chúng ta sẽ thấy mình trong tình huống được mô tả. (Tại sao?)

Việc thực hiện thuật toán này là hiệu quả nhất nếu chúng ta biểu diễn đồ thị thông qua danh sách những người kế nhiệm (hoặc danh sách những người đi trước). Chúng ta sẽ giữ nguyên số i của các bậc trước cho mỗi đỉnh $i \in V$. Tất cả num i có thể được tìm thấy với tổng độ phức tạp $\Theta(n + m)$ bằng cách duyệt qua đồ thị. Do đó, sau khi tìm được những người thừa kế i_1, i_2, \dots, i_k trên mỗi đỉnh i , chúng ta tăng thêm một $num_{i_1}, num_{i_2}, \dots, num_{i_k}$. Tiếp theo, ở mỗi bước, chúng ta chọn đỉnh j , với $num_j = 0$. Thêm nó vào cuối danh sách Z , và trừ một đơn vị num_j cho mỗi đỉnh kế tiếp j của đỉnh đã chọn. Tổng số bước sẽ là n (để biểu diễn bằng ma trận lân cận). Do đó độ phức tạp của thuật toán trở thành $\Theta(n^2)$.

Độ phức tạp có thể được giảm xuống $\Theta(m + n)$ như sau:

1) Ban đầu ta đưa vào Z tất cả các đỉnh có số bậc trước bằng 0. Chúng ta sẽ giới thiệu một con trỏ trỏ đến phần tử chưa được xét cuối cùng của Z (ở đầu con trỏ trỏ đến phần tử đầu tiên của Z).

2) Xét đỉnh i mà con trỏ trỏ tới. Đối với mỗi kế thừa của nó j , chúng ta giảm num_j đi một. Nếu bất kỳ num_j nào trở thành 0, thêm j vào cuối Z . Di chuyển con trỏ đến phần tử tiếp theo của Z và lặp lại bước 2).

Nếu đồ thị không chứa vòng lặp, sau tối đa n lần lặp lại của 2) tất cả các đỉnh của đồ thị sẽ nằm trong Z và nó sẽ được sắp xếp theo cấu trúc liên kết.

Thuật toán 2 để sắp xếp tópô không đầy đủ

Thuật toán thứ hai chúng ta sẽ xem xét có cùng độ phức tạp $\Theta(m + n)$ như thuật toán 1. Ở một mức độ nào đó, nó “đối lập” với thuật toán trước: ở mỗi bước, chúng ta sẽ tìm kiếm một đỉnh không có người thừa kế và do đó chúng ta sẽ nhận được sự sắp xếp tópô của đồ thị ngược lại. Việc triển khai rất dễ dàng và thanh lịch, sử dụng mặt trái của phép đệ quy trong chức năng thu thập thông tin theo chiều sâu:

```

void DFS(unsigned i)
{
    unsigned k;
    used[i] = 1;
    for (k = 0; k < n; k++)
        if (A[i][k] && !used[k])
            DFS(k);
    printf("%u ", i + 1);
}

```

Có thể thấy, sửa đổi duy nhất của *DFS* tiêu chuẩn của ?? là nơi in trên đầu *i*. Điều này được thực hiện sau khi tất cả các lệnh gọi đệ quy đã được thực hiện, điều này đảm bảo rằng tất cả các đỉnh có thể đạt được từ đỉnh hiện tại đã được xem và in.

Trong hàm chính, chúng ta sẽ chạy *DFS(i)* cho mỗi đỉnh *i*, mà chúng ta chưa xét đến và không có đỉnh nào trước đó. Điều kiện cuối cùng là cần thiết cho hoạt động chính xác của thuật toán và chúng ta sẽ kiểm tra nó bằng cách sử dụng array *Used[]*, được giới thiệu để đánh dấu các đỉnh đã ghé thăm. Sau đây là mã nguồn của chương trình:

Chương trình 5.16. Sắp xếp tô pô (515topsort.c)

```

#include <stdio.h>
/* Số lượng lớn có thể trong đồ thị */
#define MAXN 200
/* Số đỉnh trong đồ thị*/
const unsigned n = 5;
/* Ma trận cạnh kề của đồ thị*/
const char A[MAXN][MAXN] = {
    { 0, 1, 0, 0, 0 },
    { 0, 0, 1, 0, 1 },
    { 0, 0, 0, 1, 0 },
    { 0, 0, 0, 0, 0 },
    { 0, 0, 1, 0, 0 }
};

char used[MAXN];

/* sửa đổi DFS */
void DFS(unsigned i)

```

```

{ unsigned k;
used[i] = 1;
for (k = 0; k < n; k++)
if (A[i][k] && !used[k])
    DFS(k);
printf("%u ", i + 1);
}

int main() {
unsigned i;
/* khởi tạo */
for (i = 0; i < n; i++) used[i] = 0;
printf("Sắp xếp tô pô (theo chiều ngược lại): \n");
for (i = 0; i < n; i++)
if (!used[i])
    DFS(i);
printf("\n");
return 0;
}

```

Kết quả thực hiện chương trình:

Sắp xếp theo topo (theo thứ tự ngược lại):

4 3 5 2 1

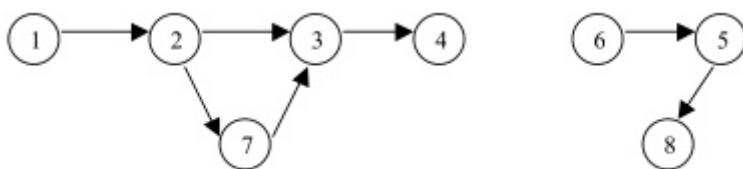
Bài tập

▷ 5.46. Hãy chứng minh Mệnh đề 5.25.

▷ 5.47. Để chứng minh rằng nếu trong bước 2 của thuật toán 1 có nhiều hơn một đỉnh không có đỉnh trước, chúng ta có thể tự do chọn bất kỳ đỉnh nào trong số chúng.

5.5.7. Sắp xếp tô pô đầy đủ

Xem xét đồ thị trong Hình 5.25. Một loại tópô cho G có thể là $Z = (1, 2, 7, 3, 4, 6, 5, 8)$ hoặc $Z = (6, 1, 2, 5, 7, 3, 8, 4)$. Sử dụng chương trình để tìm cách sắp xếp tópô đầy đủ (mà chúng ta sẽ trình bày sau), có thể tính được rằng số lượng của tất cả các danh sách Z khác nhau là 56. (Tại sao?)



Hình 5.25. Sắp xếp tô pô đầy đủ

Trong giây lát, chúng ta sẽ quay lại thuật toán 1 của 5.5.6. Khi chúng ta tìm kiếm sắp xếp theo cấu trúc liên kết, ở mỗi cấp độ, chúng ta chọn một đỉnh mà không có bậc trước, và khi có một số - chúng ta chọn bất kỳ. Các quyết định khác nhau xuất phát từ sự lựa chọn không xác định. Để tìm tập hợp tất cả các danh sách Z có thể có, chúng ta sẽ áp dụng tính năng cạn kiệt hoàn toàn. Vì vậy, khi có nhiều hơn một lựa chọn để chọn một đỉnh, chúng ta sẽ thử tất cả chúng lần lượt (rõ ràng là mỗi lựa chọn đều dẫn đến một quyết định). Trong phần triển khai bên dưới, điều này được thực hiện bởi hàm `fullTopSort()`. Khi xem một đỉnh, chúng ta sẽ xóa nó khỏi đồ thị và bắt đầu đệ quy `fullTopSort()` với đồ thị đã sửa đổi. Sau khi quay trở lại từ đệ quy, chúng ta khôi phục đồ thị và lặp lại tương tự với đỉnh có thể tiếp theo. Phần dưới cùng của đệ quy là khi danh sách Z chứa n đỉnh, tại đây chúng ta in ra nghiệm tìm được. Lược đồ của hàm đệ quy chính như sau:

```

fullTopSort(count) {
    if (count == n) {
        <Tìm thấy sắp xếp tô pô => In nó ra>;
        return;
    }
    for (<mỗi đỉnh vi không có tiền nhiệm>)
        <xóa bỏ vi của đồ thị>;
        fullTopSort(count + 1); /*đệ quy và trả về trước*/
        <khôi phục vi in ra>;
    }
}

```

Chương trình đầy đủ

Chương trình 5.17. Sắp xếp tô pô đầy đủ (516topsortf.c)

```

#include <stdio.h>
#define MAXN 200 /* số đỉnh lớn nhất*/
/* Số đỉnh của đồ thị */
const unsigned n = 8;
/* Ma trận cạnh kề */
char A[MAXN][MAXN] = {
    { 0, 1, 0, 0, 0, 0, 0, 0 },
    { 0, 0, 1, 0, 0, 0, 1, 0 },
    { 0, 0, 0, 1, 0, 0, 0, 0 },
    { 0, 0, 0, 0, 0, 0, 0, 0 },
    { 0, 0, 0, 0, 0, 0, 0, 1 },
    { 0, 0, 0, 0, 1, 0, 0, 0 },
    { 0, 0, 1, 0, 0, 0, 0, 0 },
    { 0, 0, 0, 0, 0, 0, 0, 0 }
};

char used[MAXN];
unsigned topsort[MAXN], total = 0;

void printSort(void)
{ unsigned i;
    printf("Số sắp xếp tô pô %u: ", ++total);
    for (i = 0; i < n; i++) printf("%u ", topsort[i] + 1);
    printf("\n");
}

void fullTopSort(unsigned count)
{ unsigned i, j, k, saved[MAXN];
    if (count == n) { printSort(); return; }
    /* Tìm tất cả các đỉnh không có đỉnh trước nó*/
    for (i = 0; i < n; i++) {
        if (!used[i]) {
            for (j = 0; j < n; j++)
                if (A[j][i]) break;
            if (j == n) {
                for (k = 0; k < n; k++) {
                    saved[k] = A[i][k]; A[i][k] = 0;
                }
                used[i] = 1;
                topsort[count] = i;
            }
        }
    }
}

```

```
fullTopSort(count + 1); /* hồi quy */
used[i] = 0;
for (k = 0; k < n; k++) A[i][k] = saved[k];
}
}
}
}

int main() {
unsigned i;
for (i = 0; i < n; i++) used[i] = 0;
fullTopSort(0);
return 0;
}
```

Kết quả thực hiện chương trình:

Số sắp xếp tô pô 1: 1 2 6 5 7 3 4 8

Số sắp xếp tô pô 2: 1 2 6 5 7 3 8 4

Số sắp xếp tô pô 3: 1 2 6 5 7 8 3 4

Số sắp xếp tô pô 4: 1 2 6 5 8 7 3 4

Số sắp xếp tô pô 5: 1 2 6 7 3 4 5 8

...

Số sắp xếp tô pô 49: 6 1 5 2 8 7 3 4

Số sắp xếp tô pô 50: 6 1 5 8 2 7 3 4

Số sắp xếp tô pô 51: 6 5 1 2 7 3 4 8

Số sắp xếp tô pô 52: 6 5 1 2 7 3 8 4

Số sắp xếp tô pô 53: 6 5 1 2 7 8 3 4

Số sắp xếp tô pô 54: 6 5 1 2 8 7 3 4

Số sắp xếp tô pô 55: 6 5 1 8 2 7 3 4

Số sắp xếp tô pô 56: 6 5 8 1 2 7 3 4

Bài tập

- 5.48. Tính toán phân tích số lượng sắp xếp tốpô khác nhau của đồ thị trong Hình 5.5.7.

5.5.8. Bổ sung một đồ thị xoay chiều thành một đồ thị được liên kết yếu

Bài toán: Cho đồ thị mảnh hở có hướng $G(V, E)$. Ta đang tìm đồ thị $G'(V, E')$ sao cho:

- $E \subseteq E'$
- Chỉ có một danh sách Z , là một cách sắp xếp theo topo của G' .

Nói cách khác, chúng ta đang tìm một tập hợp các cạnh để thêm vào G sao cho có một danh sách Z của G' được sắp xếp theo cấu trúc liên kết duy nhất.

Bài toán cũng có thể được đặt như sau: Bổ sung tập hợp các cạnh của một đồ thị xoay chiều có định hướng sao cho nó trở nên liên kết yếu trong khi vẫn còn xoay chiều.

Kết luận sau đây có thể được rút ra từ việc định nghĩa lại bài toán: sau khi thực hiện đóng bắc cầu trên đồ thị, ma trận lân cận A sẽ có dạng như sau:

(*) Mọi $i \neq j$ có một trong hai phần tử $A[i][j]$ và $A[j][i]$ bằng một, và còn lại - bằng không.

Thuật toán 1

Thuật toán đầu tiên chúng ta sẽ đề xuất để giải quyết bài toán như sau:

1) Chúng ta thực hiện đóng bắc cầu và, nếu có $i, j \in V, i \neq j$, mà $A[i][j] == 0$ và $A[j][i] == 0$, thì chúng ta gán $A[i][j] = 1$

2) Chúng ta thực hiện 1), trong khi (*) đúng với mọi i, j nghĩa là số các đơn vị 1 là $\frac{n^2 - n}{2}$.

Thuật toán trực quan này có độ phức tạp $\Theta(n^4)$ và không thể áp dụng trong thực tế.

Thuật toán 2

Chúng ta hãy xem xét một cách sắp xếp topô của đồ thị $Z = (v_{i1}, v_{i2}, \dots, v_{in})$. Việc hoàn thiện G' được thực hiện theo sơ đồ sau:

```

for (j = 0; j < n-1; j++)
    for (k = j+1; k < n; k++)
        if (<không tồn tại (v ij , v ik )>) /*thêm cạnh (v ij , v ik ); */
    
```

Thuật toán này rõ ràng là hiệu quả hơn Thuật toán 1 và có độ phức tạp $\Theta(n^2)$.

Bài tập

▷ 5.49. Tìm độ phức tạp của Thuật toán 1.

▷ 5.50. 2. Cho đồ thị mảnh hở $G(V, E)$. Ta tìm đồ thị $G'(V, E')$ sao cho:

- $E \subseteq E'$
- Chỉ có một danh sách Z' , đó là một cách sắp xếp theo tópô của G' .
- E' chứa một số lượng cạnh tối thiểu.

Độ phức tạp tốt nhất có thể đạt được là gì?

5.5.9. Xây dựng đồ thị của các đỉnh đã cho

Bài toán: Đã cho bậc của các đỉnh của một đồ thị vô hướng. Xây dựng đồ thị (tức là xác định tập các cạnh của nó). Có thể là bài toán không có lời giải, tức là không có đồ thị với các bậc đã cho.

Lời giải: Chúng ta sẽ giải quyết bài toán trên cơ sở quan sát sau: chúng ta hãy xem xét một đỉnh tùy ý của đồ thị, ký hiệu nó bằng i , và bậc của nó bằng $d(i)$. Rõ ràng, để có lời giải thì phải có ít nhất $d(i)$ đỉnh có bậc lớn hơn hoặc bằng một đỉnh kề với i . Chúng ta có thể nối chúng và giảm độ của chúng đi một, và đặt độ của đỉnh i là 0. Chúng ta tiếp tục càng lâu càng tốt.

chỉ còn làm rõ một cách chúng ta sẽ chọn các đỉnh (điều này không thể được thực hiện một cách ngẫu nhiên - tại sao?). Như vậy, ở mỗi bước ta sẽ chọn đỉnh có bậc cao nhất và nối lại với các đỉnh có bậc cao nhất.

Chúng ta cung cấp việc triển khai thuật toán được mô tả cho người đọc như một bài tập dễ dàng. Việc triển khai đơn giản nhất sẽ có độ phức tạp $\Theta(n^2 \cdot \log_2 n)$: n bước, trong mỗi bước chúng ta sắp xếp các mức độ. Tất nhiên, việc sắp xếp là không cần thiết, bởi vì việc sắp xếp lại, sau khi giảm các bậc đi 1, có thể được thực hiện với độ phức tạp tuyến tính (như thế nào?), Và do đó chúng ta nhận được tổng độ phức tạp $\Theta(n^2)$.

Bài tập

- ▷ 5.51. Liệu thuật toán được mô tả có hoạt động chính xác không nếu chúng ta xem xét một đồ thị là đa đồ thị?
- ▷ 5.52. Thuật toán được mô tả sẽ hoạt động chính xác nếu có các vòng lặp trong đồ thị?
- ▷ 5.53. Hãy chứng tỏ rằng thứ tự các đỉnh được coi là quan trọng.
- ▷ 5.54. Đề xuất một cách để sắp xếp lại các đỉnh bằng độ phức tạp tuyến tính.
- ▷ 5.55. Có thể giải bài toán với độ phức tạp $\Theta(m + n)$ không? Và với bài toán nhỏ hơn?

5.6. Khả năng tiếp cận và liên thông

Các bài toán về khả năng tiếp cận và liên thông được sử dụng trong máy tính, truyền thông, mạng đường bộ và các mạng khác.

Ví dụ: Một mạng máy tính được đã cho trong đó có một liên thông trực tiếp giữa một số máy tính. Nếu chúng ta trình bày mạng dưới dạng một đồ thị có định hướng, các tác vụ sau có thể được quan tâm:

- *Liên thông mạng:* Để xác định xem có một liên thông giữa hai máy tính, dù là trực tiếp hay gián tiếp, tức là đi qua các máy tính khác. Để giải quyết bài toán, cần phải kiểm tra xem đồ thị có được liên thông hay không.
- *Tìm một cụm:* Xác định số lượng máy tính được liên thông lớn nhất.
- *Ön định mạng:* Nếu tất cả các máy tính trong hệ thống được liên thông, thì bài toán tiếp theo có thể là tìm các máy tính chia sẻ hoặc liên thông chia sẻ, tức là sao cho nếu bị loại bỏ, các cặp máy tính không còn được liên thông nữa sẽ xuất hiện.

Những bài toán này và các bài toán khác sẽ được thảo luận trong phần này và được giải quyết với sự trợ giúp của các thuật toán từ lý thuyết đồ thị.

5.6.1. Các thành phần liên thông

Bài toán: Một đồ thị không định hướng được đã cho. Kiểm tra xem nó đã được liên thông chưa, tức là. cho dù có một đường đi giữa mỗi trong số hai đỉnh của nó. Nếu nó không được liên thông, hãy tìm tất cả các thành phần được liên thông của nó, tức là. tất cả các đồ thị con được liên thông tối đa của nó.

Thuật toán

Bài toán có thể được giải quyết bằng cách duyệt đồ thị, ví dụ: theo chiều sâu:

1) Chúng ta bắt đầu từ một đỉnh ngẫu nhiên và đánh dấu tất cả các đỉnh mà chúng ta coi khi thu thập thông tin là thuộc một thành phần được liên thông. Nếu nó chứa tất cả các đỉnh của đồ thị, thì nó được liên thông.

2) Nếu các đỉnh yêu cầu còn lại, điều đó có nghĩa là đồ thị không được liên thông. Chúng ta bắt đầu thu thập thông tin mới từ một đỉnh không được kiểm soát và xây dựng thành phần được liên thông thứ hai. Lặp lại bước 2) cho đến khi còn lại các đỉnh cần thiết.

Độ phức tạp của thuật toán là $\Theta(n + m)$.

Việc triển khai bao gồm một sửa đổi đơn giản của chức năng thu thập thông tin theo độ sâu. Sau đây là mã nguồn của chương trình.

Chương trình 5.18. Số thành phần liên thông (517strcon1.c)

```
#include <stdio.h>
/* Số đỉnh lớn nhất trong đồ thị */
#define MAXN 200
/* Số đỉnh trong đồ thị */
const unsigned n = 6;
/* Ma trận kè trong đồ thị */
const char A[MAXN][MAXN] = {
    { 0, 1, 1, 0, 0, 0 },
    { 1, 0, 1, 0, 0, 0 },
    { 1, 1, 0, 0, 0, 0 },
    { 0, 0, 0, 0, 1, 1 },
    { 0, 0, 0, 1, 0, 1 },
    { 0, 0, 0, 1, 1, 0 }
};
```

```

char used[MAXN];

/*Sửa đổi DFS */
void DFS(unsigned i)
{ unsigned k;
  used[i] = 1;
  printf("%u ", i + 1);
  for (k = 0; k < n; k++)
    if (A[i][k] && !used[k]) DFS(k);
}

int main()
{
  unsigned i, comp;
  /* Khởi tạo*/
  for (i = 0; i < n; i++) used[i] = 0;
  printf("\nĐây là tất cả thành phần liên thông: \n");
  comp = 0;
  for (i = 0; i < n; i++)
    if (!used[i]) {
      comp++;
      printf("{ ");
      DFS(i);
      printf("}\n");
    }
  if (l == comp)
    printf("Đồ thị liên thông.\n");
  else
    printf("Số thành phần liên thông trong đồ thị: %d \n", comp);
  return 0;
}

```

Kết quả thực hiện chương trình:

{1 2 3}
 {4 5 6}

Số thành phần được liên thông trong đồ thị: 2

Bài tập

- ▷ 5.56. Hãy triển khai một thuật toán để tìm các thành phần được liên thông của một đồ thị có trọng số chiều rộng.

5.6.2. Các thành phần liên kết mạnh trong đồ thị định hướng

Chúng ta sẽ nhớ lại định nghĩa 5.10: đồ thị định hướng $G(V, E)$ được gọi là liên thông mạnh nếu có một đường đi từ i đến j và từ j đến i , với mỗi $i \neq j, i, j \in V$. Nếu đồ thị không được liên thông mạnh, chúng ta quan tâm đến tất cả các thành phần của liên thông mạnh (tất cả các đồ thị con được liên thông mạnh nhất của G).

Các thuật toán chúng ta sẽ xem xét một lần nữa là sự thích ứng của thu thập thông tin sâu, nhưng ở đây mọi thứ không tầm thường như trong một đồ thị không định hướng.

Thuật toán 1

- 1) Ta chọn một đỉnh $i \in V$ tùy ý.
- 2) Chúng ta thực hiện $DFS(i)$ và tìm tập các đỉnh R có thể truy cập được từ i .
- 3) Chúng ta tạo thành một đồ thị "đảo ngược" $G'(V, E')$: hướng của tất cả các cạnh mà chúng "đảo ngược", tức là cạnh $(j, k) \in E'$ khi và chỉ khi $(k, j) \in E$.
- 4) Chúng ta thực hiện thu thập thông tin theo chiều sâu từ đỉnh i trong G' - vì vậy chúng ta tìm thấy tập hợp các đỉnh Q , có thể đạt được từ i trong G' (và tương ứng đạt đến i trong G).
- 5) Giao điểm của R với Q cho một thành phần liên kết mạnh.
- 6) Loại trừ thành phần này khỏi đồ thị và nếu có nhiều đỉnh hơn, hãy lặp lại bước 1).

Thuật toán được mô tả có độ phức tạp $\Theta(n.(N + m))$. Thuật toán thứ hai chúng ta sẽ trình bày có độ phức tạp $\Theta(m + n)$.

Thuật toán 2

Sửa đổi đầu tiên của chức năng thu thập thông tin theo độ sâu mà chúng ta sẽ thực hiện là đánh số các đỉnh: mỗi đỉnh sẽ nhận được một số phải lớn hơn số kế nhiệm của nó trong quá trình thu thập thông tin. Chiến lược đánh số này (được gọi là postnum trong tiếng

Anh) được thực hiện bằng cách thêm dòng sau vào hàm DFS(i) sau lệnh gọi đệ quy:

```
postnum [i] = count ++;
```

trong đó count là bộ đếm đánh số (chúng ta đã đi qua bao nhiêu đỉnh cho đến nay).

Sau đây là phần thực của thuật toán:

1) Thực hiện thu thập thông tin theo độ sâu trong G , đánh số các đỉnh theo chiến lược postnum được mô tả ở trên.

2) Chúng ta tạo thành một đồ thị $G(V, E')$, tương tự như G : sự khác biệt là hướng của tất cả các cạnh trong nó là "đảo ngược", tức là cạnh $(j, i) \in E'$ khi và chỉ khi $(i, j) \in E$.

3) Thực hiện thu thập thông tin theo độ sâu trong cột G' . Chúng ta bắt đầu tìm kiếm từ đỉnh w này, mà $postnum[w]$ có giá trị lớn nhất. Tất cả các đỉnh đạt được trong quá trình thu thập thông tin đều thuộc cùng một thành phần được liên thông mạnh. Nếu tìm kiếm không đi qua tất cả các đỉnh, thì đồ thị không được liên thông mạnh và chúng ta chọn đỉnh ban đầu thứ hai là đỉnh trong số các đỉnh không được kiểm tra mà giá trị của postnum là cao nhất. Bước này được lặp lại cho đến khi chúng ta đi xung quanh tất cả các đỉnh.

Mã nguồn của C sau đây. Điểm thú vị nhất trong đó là cách tiếp cận mà chúng ta sử dụng để thực hiện bước 2) của thuật toán. Vì sẽ cực kỳ kém hiệu quả nếu sử dụng một ma trận lân cận khác cho G' , chúng ta đã sử dụng hàm thu thập thông tin theo độ sâu thứ hai (backDFS), hàm này xem xét các cạnh của đồ thị G ở dạng "đảo ngược", như thể đang làm việc với G' . Lưu ý rằng cách tiếp cận này sẽ không thể thực hiện được với mọi bản trình bày của cột: ví dụ: với danh sách những người thừa kế.

Chương trình 5.19. Các thành phần liên thông mạnh (518strconn.c)

```
#include <stdio.h>
const N = 10; /* số đỉnh và ma trận kề của đồ thị */
int A[N][N] = {
    { 0, 1, 0, 0, 0, 0, 0, 0, 0, 0 },
    { 0, 0, 1, 1, 0, 0, 0, 0, 0, 0 },
    { 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
    { 0, 0, 0, 0, 1, 0, 1, 0, 0, 0 },
```

```

{ 0, 0, 0, 0, 0, 1, 0, 0, 0, 0 },
{ 0, 0, 1, 0, 0, 0, 0, 0, 0, 0 },
{ 0, 0, 0, 0, 0, 0, 0, 1, 0, 0 },
{ 0, 0, 0, 0, 0, 0, 0, 0, 1, 0 },
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 },
{ 0, 0, 0, 0, 0, 0, 1, 0, 0, 0 }
};

int used[N];
int postnum[N], count = 0;

/* Duy theo chiều sâu trong khi vẫn giữ nguyên số*/
void DFS(int i)
{ int j;
  used[i] = 1;
  for (j = 0; j < N; j++)
    if (!used[j] && A[i][j]) DFS(j);
  postnum[i] = count++;
}

/* Duyệt theo chiều sâu đồ thị G' */
void backDFS(int i)
{ int j;
  printf("%d ", i + 1);
  count++;
  used[i] = 1;
  for (j = 0; j < N; j++) {
    if (!used[j] && A[j][i]) {
      backDFS(j);
    }
  }
}

/* Tìm thành phần liên thông mạnh trong đồ thị*/
void strongComponents(void)
{ int i;
  for (i = 0; i < N; i++) used[i] = 0;
  while (count < N - 1) {
    for (i = 0; i < N; i++)
      if (!used[i]) DFS(i);
  }
}

```

```

for (i = 0; i < N; i++) used[i] = 0;
count = 0;
while (count < N - 1) {
    int max = -1, maxv = -1;
    for (i = 0; i < N; i++)
        if (!used[i] && postnum[i] > max) {
            max = postnum[i];
            maxv = i;
        }
    printf("{ ");
    backDFS(maxv);
    printf("}\n");
}
}

int main() {
    printf("Những thành phần liên thông mạnh là:\n");
    strongComponents();
}

```

Kết quả thực hiện chương trình:

Các thành phần có liên quan chặt chẽ trong đồ thị là:

{1 3 2 6 5 4}

{7 10 9 8}

Bài tập

▷ 5.57. Chứng minh rằng độ phức tạp của thuật toán 1 là $\Theta(n.(N + m))$.

(Gợi ý: Để chứng minh rằng trường hợp xấu nhất đối với thuật toán là một đồ thị có hướng xoay chiều với đúng $m = n.(N - 1)/2$ cạnh. Trong trường hợp này, các thành phần của liên thông mạnh là n và duyệt theo chiều sâu cho mỗi thành phần có độ phức tạp $\Theta(m + n)$.)

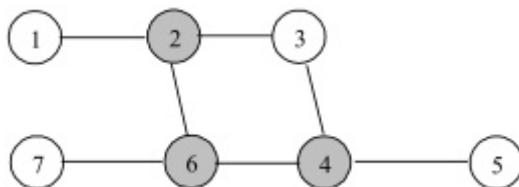
▷ 5.58. Thuật toán kiểm tra xem một đồ thị có định hướng có được *liên thông yếu* hay không và để tìm tất cả các *thành phần được liên thông yếu* của nó là gì? Lưu ý rằng, không giống như liên thông mạnh, phá vỡ đồ thị của các thành phần không giao nhau, hai thành phần được liên thông yếu khác nhau có thể bao gồm các đỉnh giống nhau. Độ phức tạp của thuật toán của bạn là gì?

5.6.3. Điểm phân chia trong một đồ thị không định hướng

Định nghĩa 5.24. Cho một đồ thị vô hướng được liên thông. Một *điểm phân chia* trong đồ thị được gọi là đỉnh, sau khi loại bỏ đỉnh đó (cũng như tất cả các cạnh liên quan đến nó) thì đồ thị không còn được liên thông.

Định nghĩa 5.25. Một đồ thị liên thông không định hướng được gọi là hai liên thông nếu không có điểm phân chia. Điều này cũng có nghĩa là nó vẫn được liên thông sau khi loại bỏ bất kỳ đỉnh nào.

Đối với đồ thị liên thông trong Hình 5.26 tách các đỉnh 2, 4 và 6. Ví dụ, nếu chúng ta loại bỏ đỉnh 6, chúng ta nhận được hai thành phần của liên thông: $\{1, 2, 3, 4, 5\}$ và $\{7\}$.



Hình 5.26. Điểm phân tách trong đồ thị

Nếu chúng ta sử dụng trực tiếp định nghĩa, chúng ta có thể đạt được độ phức tạp $\Theta(n \cdot (M + n))$: chúng ta sẽ sử dụng thuật toán để kiểm tra xem một đồ thị không định hướng có được liên thông hay không. Mỗi đỉnh, sau khi loại bỏ mà đồ thị không còn được liên thông, là một *điểm phân chia*.

Thuật toán trên rất dễ thực hiện, nhưng không phải là hiệu quả nhất.

Mệnh đề 5.2. *Đỉnh $k \in V$ là điểm phân chia khi và chỉ khi có hai đỉnh khác i và j ($i, j \in V$) mà mỗi đỉnh giữa chúng đi qua k .*

Mệnh đề 5.3. *Gọi $G(V, E)$ là một đồ thị không có hướng được liên thông và $T(V, D)$ là một cây bao trùm của G , được xây dựng bằng cách duyệt theo chiều sâu. Đỉnh $k \in V$ sẽ là điểm phân chia nếu và chỉ khi tồn tại các đỉnh $i, j \in V$ sao cho thỏa mãn các điều kiện đồng thời:*

- 1) $(k, i) \in D$
- 2) $j \neq k$
- 3) j không phải là kế vị của i trong T
- 4) $\text{lowest}[i] \geq \text{prenum}[k]$ trong đó:
 - $\text{prenum}[k]$ là số nhận được bởi đỉnh k , khi đánh số sơ bộ của các đỉnh (tức là số của các đỉnh được thu thập thông tin nhận được ở đầu hàm thu thập thông tin trước khi gọi đệ quy)
 - $\text{lowest}[i]$ nhận được ít nhất từ:
 - a) $\text{prenum}[i]$
 - b) $\text{prenum}[w]$ cho mỗi đỉnh w sao cho $(i, w) \in E$ và $(i, w) \in D$.
 - c) $\text{lowest}[w]$ cho mỗi người thừa kế w của i trong D .

Dựa trên Mệnh đề 2, chúng ta sẽ tìm thấy các điểm phân chia trong một đồ thị không định hướng:

Thuật toán

1) Chúng ta thực hiện thu thập thông tin theo chiều sâu từ bất kỳ đỉnh nào của G . Gọi T là cây bao phủ thu được kết quả là. Đối với mỗi đỉnh i với $\text{prenum}[i]$, chúng ta đã chỉ ra số prenum của đỉnh thu được trong quá trình thu thập thông tin.

2) Chúng ta đi xung quanh cây T trong LDK (xem). Đối với mỗi đỉnh i , mà chúng ta xem xét, chúng ta tính $\text{lowest}[i]$ theo cách được mô tả trong Mệnh đề 2.

3) Các điểm phân chia được xác định như sau:

3.1) Gốc của cây T là điểm phân chia khi và chỉ khi có nhiều hơn một người thừa kế.

3.2) Đỉnh i khác với gốc của T là điểm phân chia nếu và chỉ khi i có một kế trực tiếp x mà đỉnh $\text{lowest}[x] \geq \text{prenum}[i]$ đang giữ.

Độ phức tạp của thuật toán này, khi trình bày cột với danh sách các phần tử kế tiếp, là $\Theta(n + m)$. (Tại sao?) Tuy nhiên, nhận thức của chúng ta là $\Theta(n^2)$.

Trong cách triển khai được đề xuất dưới đây, chúng ta đã sử dụng biểu diễn của đồ thị với ma trận lân cận. Chúng ta sẽ không sử dụng một mảng bổ sung cho cây phủ T : để chỉ ra rằng một cạnh tham gia vào T , chúng ta sẽ sử dụng giá trị bổ sung $+2$ trong ma trận lân cận $A[][]$

Chương trình 5.20. Các điểm phân tách trong đồ thị (519artic.c)

```
#include <stdio.h>
#define MAXN 150 /* Số đỉnh cực đại trong đồ thị */
/* Số đỉnh đồ thị */
const unsigned n = 7;
/* Ma trận kè */
char A[MAXN][MAXN] = {
    { 0, 1, 0, 0, 0, 0, 0 },
    { 1, 0, 1, 0, 0, 1, 0 },
    { 0, 1, 0, 1, 0, 1, 0 },
    { 0, 0, 1, 0, 1, 1, 0 },
    { 0, 0, 0, 1, 0, 0, 0 },
    { 0, 1, 1, 1, 0, 0, 1 },
    { 0, 0, 0, 0, 0, 1, 0 }
};

unsigned prenum[MAXN], lowest[MAXN], cN;
unsigned min(unsigned a, unsigned b) { return (a < b) ? a : b; }

void DFS(unsigned i)
{ unsigned j;
    prenum[i] = ++cN;
    for (j = 0; j < n; j++)
        if (A[i][j] && !prenum[j]) {
            A[i][j] = 2; /* Xây dựng cây phủ T */
            DFS(j);
        }
}

/* Duyệt cây theo postorder */
void postOrder(unsigned i)
{ unsigned j;
    for (j = 0; j < n; j++)
        if (2 == A[i][j]) postOrder(j);
    lowest[i] = prenum[i];
    for (j = 0; j < n; j++)
        if (1 == A[i][j]) lowest[i] = min(lowest[i], prenum[j]);
    for (j = 0; j < n; j++)
        if (2 == A[i][j]) lowest[i] = min(lowest[i], lowest[j]);
}
```

```

void findArticPoints(void)
{ unsigned artPoints[MAXN], i, j, count;
  for (i = 0; i < n; i++) {
    prenum[i] = 0; lowest[i] = 0; artPoints[i] = 0;
  }
  cN = 0;
  DFS(0);
  for (i = 0; i < n; i++) {
    if (0 == prenum[i]) {
      printf("Đồ thị không liên thông - \n");
      return;
    }
  }
  postOrder(0);

  /* kiểm tra 3.1) */
  count = 0;
  for (i = 0; i < n; i++)
    if (2 == A[0][i]) count++;
  if (count > 1) artPoints[0] = 1;

  /* sử dụng bước 3.2) */
  for (i = 1; i < n; i++) {
    for (j = 0; j < n; j++)
      if (2 == A[i][j] && lowest[j] >= prenum[i]) break;
    if (j < n) artPoints[i] = 1;
  }

  printf("Những điểm tách trong đồ thị là:\n");
  for (i = 0; i < n; i++)
    if (artPoints[i]) printf("%u ", i + 1);
  printf("\n");
}

int main(){
  findArticPoints();
  return 0;
}

```

Kết quả thực hiện chương trình:

Các điểm phân chia trong cột là:

2 4 6

Bài tập

▷ 5.59. Chứng minh mệnh đề 1 và mệnh đề 2.

5.6.4. *k-liên thông của đồ thị không định hướng*

Định nghĩa 5.26. Đồ thị không định hướng được gọi là *k-liên thông đối với các đỉnh* nếu nó được liên thông và vẫn liên thông sau khi loại bỏ $k - 1$ ngẫu nhiên đỉnh.

Định nghĩa 5.27. Đồ thị không định hướng được gọi là *k-liên thông đối với các cạnh* nếu nó được liên thông và vẫn liên thông sau khi loại bỏ $k - 1$ ngẫu nhiên cạnh.

Cho đồ thị $G(V, E)$. Chúng ta sẽ xem xét các bài toán sau:

- Tìm k nhỏ nhất mà trong đó G không được *k-liên thông* theo đỉnh.
- Tìm k nhỏ nhất mà G không được *k-liên thông* theo các cạnh.

Chúng ta sẽ phác thảo cách hai bài toán trên có thể được quyết định bằng cách sử dụng bài toán của luồng tối đa.

k-liên thông theo các cạnh

Tầm quan trọng của luồng tối đa (có dung lượng của tất cả các bộ phận cạnh) giữa nguồn i và người tiêu dùng j bằng số cạnh tối thiểu cần thiết để phân chia hai đỉnh để chúng vẫn ở trong các thành phần liên thông khác nhau (tại sao?). Do đó, độ k tối thiểu bằng với giá trị tối thiểu của các luồng tối đa giữa đỉnh bất kỳ i và đỉnh $n - 1$ đỉnh còn lại của đồ thị.

k-liên thông về các đỉnh

Định lý 5.8 (Menger). *Đồ thị k-liên thông theo đỉnh khi và chỉ khi mỗi cặp các đỉnh được liên thông bởi k đường không cắt nhau (không trùng các đỉnh trung gian).*

Thuật toán để kiểm tra *k-liên thông* về đỉnh:

- Xây dựng số lượng $G'(V', E')$ với thuộc tính sau: Trong mỗi tập hợp các đường không giao nhau theo đỉnh trong G tương ứng với tập hợp các đường không giao nhau theo cạnh trong G' .

Việc xây dựng được thực hiện như sau:

- Trên mỗi đỉnh $i \in V$ tương ứng hai đỉnh $i_1 \in V'$ và $i_2 \in V'$ và cạnh $(i_1, i_2) \in E'$.
 - Trên mỗi cạnh $(i, j) \in E$ tương ứng các cạnh $(i_1, j_2) \in E'$ và $(i_2, j_1) \in E'$.
- Thông qua một luồng tối đa trong G' , chúng ta kiểm tra xem có k biết đường dẫn không giao nhau nào (theo cạnh trong chúng các cạnh liên quan):
 - Nếu chúng tồn tại, thì suy ra đồ thị k -liên kết: Thuộc tính của những con đường không được chuyển từ các cạnh G' vào các đỉnh trong G . (Tại sao?) Áp dụng định lý của Meninger cho những đường không giao nhau trên G và chúng ta nhận được kết quả tìm kiếm.
 - Nếu chúng không tồn tại, đồ thị không là k -liên thông.

Bài tập

- ▷ 5.60. Hãy chứng minh rằng tính chất của những con đường không giao nhau chuyển từ các cạnh trong G' vào các đỉnh trong G .
- ▷ 5.61. Sửa đổi thuật toán để kiểm tra xem đồ thị có k -liên thông với các đỉnh sao cho với tìm k -liên thông tối đa của đồ thị.

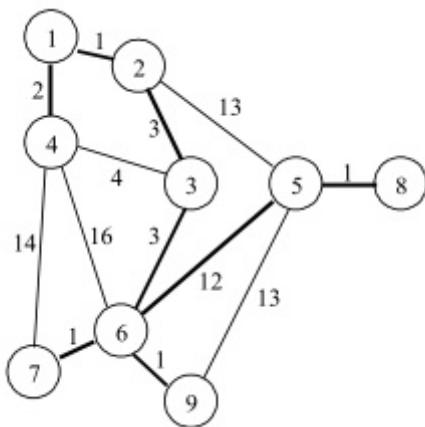
5.7. Các tập hợp con tối ưu và các tâm của đồ thị

5.7.1. Cây bao phủ tối thiểu

Để tưởng tượng một nhóm các hòn đảo, chúng ta muốn kết nối với những cây cầu theo cách mà nó có thể đạt được từ mọi hòn đảo đến nhau trong nhóm. Vì việc xây dựng cầu có liên quan đến một số chi phí nhất định, mục tiêu chúng ta sẽ là vô số cây cầu mà chúng ta xây dựng, có giá tối thiểu. Chúng ta sẽ cho rằng đối với mỗi cặp đảo chúng ta biết nếu nó có thể được kết nối trực tiếp với một cây

cầu và chi phí bao nhiêu.

Trình bày theo lý thuyết về đồ thị, bài toán trên sẽ trông như thế này: một số lượng không bị kéo $G(V, E)$ trong đó đỉnh của tập hợp V là độc lập và cạnh (i, j) tồn tại sao cho có thể xây dựng một cây đường dẫn giữa các đỉnh độc lập i và j . Trọng số $f(i, j)$ xác định giá cả có thể của nó. Tìm bao phủ (bao phủ) $T(V, D)$ của G với trọng số tối thiểu của tổng trọng số các cạnh có trong D .



Hình 5.27. Cây bao phủ nhỏ nhất trong đồ thị

Cây bao trùm T với tính chất như vậy được gọi là *cây phủ tối thiểu*. Chúng ta xem xét hai thuật toán phổ biến (xem Chương 9) giải bài toán đã cho.

- Thuật toán Kruskal

Xem xét đồ thị không hướng trọng số $G(V, E)$.

Thuật toán của Kruskal để tìm cây phủ tối thiểu

- 1) Tạo n bộ, như trong tập hợp thứ i đặt đặt đỉnh thứ i của đồ thị.
- 2) Sắp xếp các cạnh của cột theo thứ tự tăng dần (tính theo trọng số của chúng).
- 3) Tạo một cây rỗng $T(V, \emptyset)$. Sau khi hoàn thành thuật toán T tìm được cây bao phủ tối thiểu.

4) Thực hiện $(n - 1)$ lần. Thêm cạnh $(i, j) \in E$ cho T , sao cho hoàn thành:

- Trọng số $f(i, j)$ có thể nhỏ nhất có thể (có một danh sách được sắp xếp, trọng lượng của các cạnh của đồ thị)
- Các đỉnh i và j được đặt trong các bộ khác nhau, sau mỗi bổ sung (i, j) lấy hợp các tập hợp, trong đó có i và j .

Lưu ý: Mỗi cạnh được nhìn thấy nhiều nhất một lần. Thật vậy, nếu bất kỳ bước nào và j nào trong cùng một bộ, họ sẽ vẫn còn "cùng nhau" tất cả các cách: thuật toán chỉ đơn vị nhưng không phá vỡ. Do đó, nếu một cạnh một lần là "không phù hợp", nó sẽ là thích ứng và mỗi lần lặp tiếp theo của bước 4). Quá trình xây dựng cây kết thúc tại việc tích hợp vào $n - 1$ (mỗi chu trình tiếp theo sẽ đóng lại, tại sao?).

Xem xét cách thuật toán cho ví dụ trong Hình 5.7. Ban đầu, chín đỉnh cột được đặt trong chín bộ khác nhau (các thành phần):

[{1}](#) [{2}](#) [{3}](#) [{4}](#) [{5}](#) [{6}](#) [{7}](#) [{8}](#) [{9}](#)

Danh sách các cạnh được sắp xếp có được trong bước 2) là như sau:

[\(1, 2\) = 1](#); [\(5, 8\) = 1](#); [\(7, 6\) = 1](#); [\(6, 9\) = 1](#); [\(1, 4\) = 2](#); [\(2, 3\) = 3](#); [\(3, 6\) = 3](#); [\(3, 4\) = 4](#); [\(5, 6\) = 12](#); [\(5, 9\) = 13](#); [\(2, 5\) = 13](#); [\(4, 7\) = 14](#); [\(4, 6\) = 14](#);

Chúng ta chọn cạnh đầu tiên $(1, 2)$ và kết hợp các tập hợp, trong đó nằm là đỉnh 1 và 2. Chúng ta có được [{1, 2}](#) [{3}](#) [{4}](#) [{5}](#) [{6}](#) [{7}](#) [{8}](#) [{9}](#).

Chúng ta chọn cạnh lớn nhất tiếp theo $(5, 8)$. Các tập hợp là: [{1, 2}](#) [{3}](#) [{4}](#) [{5, 8}](#) [{6}](#) [{7}](#) [{9}](#).

Bên cạnh cây che phủ, chúng ta liên tiếp bao gồm $(7, 6)$, $(6, 9)$, $(1, 4)$, $(2, 3)$, $(3, 6)$ và chúng ta nhận được: [{1, 2, 3, 4, 6, 7, 9}](#) [{5, 8}](#).

Chúng ta bỏ qua cạnh $(3, 4)$, vì các đỉnh 3 và 4 đã nằm trong cùng một tập hợp. Chúng ta lấy cạnh tiếp theo $(5, 6)$, nơi tất cả các đỉnh của đồ thị thuộc một tập hợp chung. Lớp gỗ che phủ tối thiểu được xây dựng và bao gồm các cạnh $(1, 2)$, $(1, 4)$, $(2, 3)$, $(3, 6)$, $(6, 7)$, $(6, 9)$, $(6, 5)$), $(5, 8)$.

Hãy tính độ phức tạp của thuật toán. Các bước cần thiết để khởi tạo bộ là $\Theta(n)$. Hơn nữa, độ phức tạp được xác định bởi hai điều. Đầu tiên là cách chúng ta sẽ thực hiện việc sắp xếp ở bước 2). Nếu chúng ta sử dụng sắp xếp theo hình chóp (xem ??), Độ phức tạp của sắp xếp trong trường hợp xấu nhất sẽ là $\Theta(m \log_2 m)$. Yếu tố quyết định thứ hai là cách xác minh rằng hai đỉnh thuộc cùng một tập hợp sẽ được thực hiện (và việc hợp nhất các tập hợp trong mỗi lựa chọn cạnh ở bước 4). Chúng ta sẽ sử dụng các thành phần của kết nối (những thành phần này đã được thảo luận khi xem xét các cách trình bày đồ thị - 5.2.). Đối với mỗi thành phần của kết nối, chúng ta xây dựng một cây với gốc của bất kỳ phần nào trên cùng của nó. Khi chúng ta thêm một cạnh mới vào đồ thị, chúng ta sẽ kết hợp hai cây tương ứng với hai thành phần kết nối (ví dụ, thêm một cạnh trong đó gốc của cây này kế thừa gốc của cây kia). Việc kiểm tra xem hai đỉnh có trong cùng một thành phần kết nối hay không được thực hiện bằng cách so sánh các gốc của cây tương ứng với các thành phần này (nghĩa là các rẽ này có phải là cùng một đỉnh hay không). Tiếp cận gốc của cây (bắt đầu từ một chiếc lá) có log độ phức tạp $\Theta(\log_2 n)$ và vì chúng ta thực hiện $n - 1$ bước nên tổng độ phức tạp của thuật toán sẽ là: $\Theta(m \log_2 m + n \log_2 n)$.

Trong việc thực hiện C, đồ thị được biểu diễn bằng danh sách các cạnh trong mảng $S[]$ với M phần tử (bằng số cạnh trong đồ thị). $S[i].x$, $S[i].y$, $S[i].Weight$ cho thấy rằng cạnh $(S[i].x, S[i].y)$ có trọng lượng $S[i].weight$. Với sắp xếp hàm (A , \emptyset , M), chúng ta sắp xếp các cạnh của đồ thị theo trọng số của chúng theo thứ tự tăng dần. Chúng ta sẽ triển khai các phép toán với các thành phần của kết nối và các cây tương ứng của chúng như sau: Chúng ta sẽ giới thiệu một mảng $prev[]$, trong đó $prev[i]$ là một đỉnh của cùng một tập hợp mà i thuộc về, và là đỉnh của nó (trong bộ này, không phải trong $D!$). Nếu đỉnh i là gốc của cây tương ứng, thì $prev[i] == \emptyset$. Sự kết hợp của hai tập hợp được thực hiện bằng cách thực thi $prev[r2] = r1$, trong đó $r1$ và $r2$ là gốc của các cây mà chúng ta hợp nhất. Xác định gốc của cây có đỉnh i như sau:

```
root = i;
while (NO_PARENT != prev[root]) root = prev[root];
```

Ngoài ra, sau mỗi lần tìm kiếm như vậy, chúng ta sẽ thực hiện co

đường. Nó được thực hiện để giảm tổng chiều sâu của cây với chi phí thu thập thêm một lần thu thập thông tin từ i đến gốc. Sau mỗi lần tìm kiếm, các phần tử trên đường dẫn từ i đến gốc sẽ kế thừa gốc:

```
while ( $i \neq$  root) {
    savei = i;
     $i = prev[i];$ 
    prev[savei] = root;
}
```

Với nhiều yếu tố hơn, chúng ta có thể cải thiện nhiều hơn nữa bằng cách chia sẻ khả năng cân bằng gỗ và co ngót đường. Tuy nhiên, việc cân bằng đòi hỏi một lượng bộ nhớ bổ sung theo thứ tự của tổng số n phần tử của tập hợp, và ở các giá trị thấp hơn của n không mang lại nhiều lợi ích so với việc thu nhỏ đường mà không cân bằng. Do đó, trong quá trình thực hiện thuật toán Kruskal, chúng ta sẽ sử dụng hàm trên để thu nhỏ đường không cân bằng. Rõ ràng, số lượng các phép toán tìm kiếm và phép nối tỷ lệ với tổng số n phần tử của các tập hợp. Có thể thấy độ phức tạp của các cách tiếp cận khác nhau trong trường hợp này từ Bảng 5.3, trong đó với $\log * n$, chúng ta đã biểu thị số ứng dụng của hàm logarit là n , cần thiết để lấy 0. Các dữ liệu đầu vào được sử dụng trong mã nguồn

Số phép tính cơ sở tìm kiếm và hợp nhất	Cân bằng trên cây	Rút ngắn đường đi
n^2	KHÔNG	KHÔNG
$n.\log n$	CÓ	KHÔNG
$n.\log * n$	CÓ	CÓ

Bảng 5.3. Số lượng phép toán cơ sở tìm kiếm và hợp nhất các cây với n phần tử trong các chiến lược khác nhau

dưới đây là cho đồ thị trong Hình 5.27.

Chương trình 5.21. Cây bao phủ nhỏ nhất trong đồ thị (520kruskal.c)

```
#include <stdio.h>
/*Số đỉnh tối đa có thể trong đồ thị */
```

```
#define MAXN 200
#define NO_PARENT (unsigned)(-1)
/* Số cạnh tối đa có thể trong đồ thị */
#define MAXM 2000
const unsigned n = 9; /* Số đỉnh trong đồ thị */
const unsigned m = 14; /* Số cạnh trong đồ thị */
struct arc {
    unsigned i, j;
    int f;
};
/* Danh sách cạnh đồ thị và trọng số của chúng*/
struct arc S[MAXM] = {
    { 1, 2, 1},
    { 1, 4, 2},
    { 2, 3, 3},
    { 2, 5, 13},
    { 3, 4, 4},
    { 3, 6, 3},
    { 4, 6, 16},
    { 4, 7, 14},
    { 5, 6, 12},
    { 5, 8, 1},
    { 5, 9, 13},
    { 6, 7, 1},
    { 6, 9, 1}
};

int prev[MAXN + 1];

int getRoot(int i)
{ int root, savei;
    /* Tìm gốc của cây */
    root = i;
    while (NO_PARENT != prev[root]) root = prev[root];
    /* Rút ngắn đường */
    while (i != root) {
        savei = i;
        i = prev[i];
        prev[savei] = root;
    }
}
```

```

return root;
}

void kruskal(void)
{
    int MST = 0;
    unsigned i, j;
    /* Sắp xếp danh sách với cạnh trọng số tăng dần*/
    sort(S, 0, m);

    printf("Các cạnh tham gia vào cây bao phủ nhỏ nhất:\n");
    for (i = 0; i < m; i++) {
        int r1 = getRoot(S[i].i);
        int r2 = getRoot(S[i].j);
        if (r1 != r2) {
            printf("(%u,%u)", S[i].i, S[i].j);
            MST += S[i].f;
            prev[r2] = r1;
        }
    }
    printf("\nGiá cây bao phủ nhỏ nhất là %d.\n", MST);
}

int main() {
    unsigned i;
    for (i = 0; i < n + 1; i++) prev[i] = NO_PARENT;
    kruskal();
    return 0;
}

```

Kết quả thực hiện chương trình:

Các cạnh tham gia vào cây bao phủ nhỏ nhất:

(1.2) (5.8) (6.7) (6.9) (1.4) (2.3) (3.6)

Giá cây bao phủ nhỏ nhất là 24.

Bài tập

- ▷ 5.62. Hãy chứng minh rằng thuật toán của Kruskal hoạt động chính xác.
- ▷ 5.63. Thuật toán của Kruskal có hoạt động đối với một đồ thị

không liên thông không? Nếu vậy, tiêu chí dừng việc bổ sung thêm cạnh mới là gì?

▷ **5.64.** Thuật toán của Kruskal có hoạt động đối với một đồ thị có định hướng không?

- Thuật toán của Prim

Một giải pháp thay thế cho thuật toán của Kruskal để tìm cây bao trùm nhỏ nhất là thuật toán Prim.

Thuật toán Prim

1) Chúng ta bắt đầu xây dựng cây bao trùm nhỏ nhất từ đỉnh s bất kỳ: ban đầu cây sẽ là $T(H, D)$, $H = s$, $D = \emptyset$.

2) Lặp lại $n - 1$ lần:

2.1) Chúng ta chọn cạnh $(i, j) \in E$, sao cho:

- $i \in H, j \in V \setminus H$;
- $f(i, j)$ là cực tiểu.

2.2) Thêm đỉnh j vào H và cạnh (i, j) vào D .

Đây là cách thuật toán được áp dụng cho ví dụ trong Hình 5.27: Chúng ta chọn một đỉnh ban đầu tùy ý, ví dụ 1. Cạnh nhỏ nhất nối nó với một đỉnh không tham gia vào H là $(1, 2)$. Thêm đỉnh 2 vào H và cạnh $(1, 2)$ vào D , thành D . Cạnh nhỏ nhất tiếp theo nối đỉnh H với đỉnh không thuộc H là $(1, 4)$. Tiếp theo, các sườn được chọn sẽ là $(2, 3), (3, 6), (6, 7), (6, 9), (6, 5)$ và $(5, 8)$, với gỗ phủ tối thiểu là được xây dựng.

Hãy triển khai thuật toán được mô tả. Bước cần được chỉ rõ thêm là 2.1). Nếu chúng ta áp dụng phương pháp tiếp cận trực tiếp, trong đó chúng ta luôn kiểm tra và duy trì khoảng cách nhỏ nhất giữa các cặp đỉnh thuộc và không thuộc H , chúng ta sẽ thu được thuật toán có độ phức tạp $\Theta(n^3)$. Để tránh kiểm tra nhiều hơn n lần, chúng ta sẽ giới thiệu thêm một mảng $T[]$ - trong đó chúng ta sẽ giữ khoảng cách ngắn nhất đến các đỉnh (không nằm trên H) chưa được xem xét, tức là khoảng cách từ cây đến mỗi đỉnh ngoài cây:

- Giả sử rằng đỉnh bắt đầu là s , chúng ta khởi tạo ở đầu:
 - $T[j] = A[s][j], j = 1, 2, \dots, n$, cho mỗi $(s, j) \in E$
 - $T[j] = MAX_VALUE$, ngược lại.

- Ở mỗi bước:

- Ta thấy rằng j mà $T[j]$ có giá trị nhỏ nhất.
- Với mỗi $k \in H$ ta thực hiện $T[k] = \min(T[k], A[j][k])$.

Sau đây là mã nguồn của chương trình thực hiện thuật toán được mô tả.

Chương trình 5.22. Thuật toán Prim tìm cây bao trùm (521prim.c)

```
#include <stdio.h>
/* Số đỉnh lớn nhất có thể */
#define MAXN 150
#define MAX_VALUE 10000
/* Số đỉnh của đồ thị */
const unsigned n = 9;
/* Ma trận trọng số của đồ thị */
const int A[MAXN][MAXN] = {
    { 0, 1, 0, 2, 0, 0, 0, 0, 0 },
    { 1, 0, 3, 0, 13, 0, 0, 0, 0 },
    { 0, 3, 0, 4, 0, 3, 0, 0, 0 },
    { 2, 0, 4, 0, 0, 16, 14, 0, 0 },
    { 0, 13, 0, 0, 0, 12, 0, 1, 13 },
    { 0, 0, 3, 16, 12, 0, 1, 0, 1 },
    { 0, 0, 0, 14, 0, 1, 0, 0, 0 },
    { 0, 0, 0, 0, 1, 0, 0, 0, 0 },
    { 0, 0, 0, 0, 13, 1, 0, 0, 0 }
};

char used[MAXN];
unsigned prev[MAXN];
int T[MAXN];

void prim(void)
{ int MST = 0; /* Ở đây tích lũy giá của cây bao phủ tối thiểu */
  unsigned i, k;
  /* Khởi tạo */
  for (i = 0; i < n; i++) { used[i] = 0; prev[i] = 0; }
  used[0] = 1; /* Chọn đỉnh bắt đầu bất kỳ */
  for (i = 0; i < n; i++)
    T[i] = (A[0][i]) ? A[0][i] : MAX_VALUE;
  for (k = 0; k < n - 1; k++) {
    /* Tìm cạnh nhỏ nhất tiếp theo */
```

```

int minDist = MAX_VALUE, j = -1;
for (i = 0; i < n; i++)
if (!used[i])
    if (T[i] < minDist) {
        minDist = T[i];
        j = i;
    }
used[j] = 1;
printf("(%u,%u)", prev[j] + 1, j + 1);
MST += minDist;
for (i = 0; i < n; i++)
if (!used[i] && A[j][i]) {
    if (T[i] > A[j][i]) {
        T[i] = A[j][i];
        /* lưu cạnh phía trước để in được cạnh tối thiểu tiếp theo*/
        prev[i] = j;
    }
}
printf("\nGiá của cây phủ nhỏ nhất là %d.\n", MST);
printf("\n");
}

int main() {
    prim();
    return 0;
}

```

Kết quả thực hiện chương trình:

(1.2) (1.4) (2.3) (3.6) (6.7) (6.9) (6.5) (6.5)

Giá của cây bao phủ tối thiểu là 24.

Dễ dàng nhận thấy rằng độ phức tạp của thuật toán Prim được thực hiện theo cách này là bậc hai về số đỉnh của đồ thị (Tại sao?).

Với việc lựa chọn cấu trúc dữ liệu cẩn thận hơn (ví dụ: nếu sử dụng cấu trúc hình chóp [Cormen, Leiserson, Rivest-1997]), độ phức tạp của thuật toán Prim có thể đạt đến $\Theta(m + n \log_2 n)$.

Đối với trường hợp đặc biệt của đồ thị thưa, tức là khi $m \in O(n)$, thuật toán Prim thậm chí còn hiệu quả hơn, với độ phức

tập $\Theta(m.\beta(m, n))$ và thậm chí $\Theta(m.\log_2(\beta(m, n)))$, ở đâu ([Cormen, Leiserson, Rivest-1997] [Fredman, Tarjan-1987]):

$$\beta(m, n) = \min i \text{ sao cho } \underbrace{\log(\log(\dots\log(n)))}_{i \text{ lần}} < \frac{m}{n}$$

Bài tập

- ▷ 5.65. Chứng minh rằng thuật toán Prim hoạt động chính xác.
- ▷ 5.66. Thuật toán của Prim có hoạt động đối với một đồ thị không liên quan không?
- ▷ 5.67. Thuật toán của Prim có hoạt động đối với một đồ thị có định hướng không?
- ▷ 5.68. Những tối ưu hóa nào có thể được áp dụng cho các thuật toán của Prim và Kruskal cho trường hợp trọng số của các cạnh của đồ thị nằm trong khoảng $[1, n]$?

- Cây bao phủ tối thiểu một phần

Chúng ta sẽ xem xét sửa đổi Bài toán xây dựng Cây bao phủ nhỏ nhất.

Định nghĩa 5.28. Một đồ thị $G(V, E)$ đã cho. Với k cho trước, $k \leq n$ cây bao phủ một phần cực tiểu $T_k(V_k, D)$ được gọi là cây chứa đúng k đỉnh, sao cho:

- $V_k \subseteq V$.
- $D \subseteq E$.
- tổng trọng số của các cạnh liên quan đến D là nhỏ nhất.

Thuật toán tìm cây bao phủ nhỏ nhất một phần:

Đối với mỗi đỉnh của đồ thị, chúng ta sẽ cố gắng xây dựng một cây bao phủ tối thiểu tương ứng theo thuật toán Prim, nhưng chứa đúng k đỉnh. Chúng ta sẽ ngắt thuật toán tại thời điểm $k - 1$ cạnh đã được thêm vào (tức là $k - 1$ bước đã được thực hiện). Theo cách này, chúng ta sẽ lấy n cây (mỗi đỉnh một cây), từ đó chúng ta sẽ chọn ra cây có giá thấp nhất.

Chúng ta cung cấp cho người đọc việc triển khai sự thích ứng của thuật toán Prim.

Bài tập

- ▷ 5.69. Hãy chứng minh thuật toán được đề xuất để xây dựng cây bao phủ tối thiểu một phần trên cơ sở thuật toán Prim là đúng.
- ▷ 5.70. Hãy triển khai sửa đổi đề xuất của thuật toán Prim.

5.7.2.

5.7.2. Tập đỉnh độc lập

Chúng ta sẽ nhắc lại hai định nghĩa từ đầu chương: Một đồ thị $G(V, E)$ được gọi là đầy đủ nếu tồn tại một cạnh (i, j) với mọi $i, j \in V$. Số lần click là số đỉnh trong đồ thị con đầy đủ lớn nhất $G'(V', E')$ của G . Phần nghịch đảo của đồ thị đầy đủ là đồ thị trống - không chứa cạnh. Chúng ta sẽ xem xét hai bài toán và xác định một khái niệm, trái ngược với số lần click.

Bài toán 1: Một nhóm gồm n người được đã cho, trong đó có hai người là bạn hoặc không phải là bạn của nhau. Chọn số lượng tối đa để mọi người trong nhóm đã chọn là bạn bè.

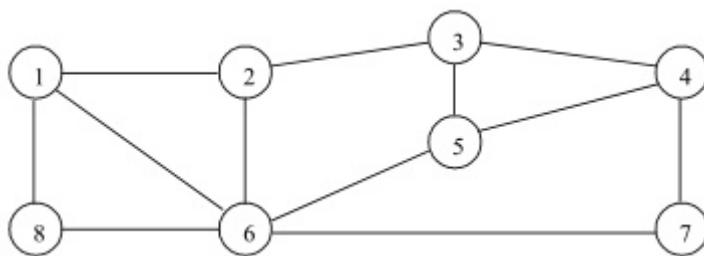
Bài toán 2: Một nhóm gồm n người được đã cho, trong đó có hai người là hoặc không phải là kẻ thù của nhau. Chọn số người tối đa để không có hai người trong nhóm được chọn là kẻ thù.

Chúng ta sẽ giải quyết bài toán thứ hai. Câu thứ nhất có thể được giải quyết với sự trợ giúp của câu thứ hai (chúng ta để nó như một bài tập để người đọc tìm ra cách, sau khi làm quen với tài liệu thêm).

Định nghĩa 5.29. Một đồ thị vô hướng $G(V, E)$ được cho. Tập hợp $H, H \subseteq V$ được gọi là *tập các đỉnh độc lập* nếu nó không chứa các đỉnh liền kề.

Định nghĩa 5.30. Số phần tử của tập đỉnh độc lập có số đỉnh lớn nhất được gọi là *số độc lập* của G .

Nói cách khác, số lượng độc lập của một đồ thị vô hướng là số đỉnh trong số các đỉnh lớn nhất của nó trong một đồ thị con trống.



Hình 5.28. Đồ thị vô hướng

Ví dụ, cho đồ thị trong Hình 5.28 các tập đỉnh độc lập là $\{7, 8, 2\}$, $\{3, 1\}$, $\{7, 8, 2, 5\}$. Số độc lập của đồ thị là 4.

- Tập hợp độc lập tối đa

Định nghĩa 5.31. Một tập độc lập H trong đồ thị vô hướng được gọi là *cực đại* nếu không có tập độc lập H' nào khác chứa H như một tập con thực sự.

Ví dụ, trong cột của Hình 5.28, tập độc lập lớn nhất là $H_1 = \{7, 8, 2, 5\}$, trong khi $H_2 = \{7, 8, 2\}$ thì không, vì nó được chứa trong H_1 . Các tập hợp độc lập tối đa cũng là $\{1, 3, 7\}$ và $\{4, 6\}$.

Thuật toán tìm tất cả các tập độc lập cực đại

Chúng ta hãy xem xét đồ thị $G(V, E)$. Đầu tiên chúng ta hãy xem xét một thuật toán để tìm một tập độc lập tối đa T :

- 1) Cho S và T là các tập rỗng lúc đầu.
- 2) Ta chọn một đỉnh tùy ý từ đồ thị $i \in V, i \notin S \cup T$.
- 3) Thêm i vào T . Trong S thêm tất cả các đỉnh kề với i .
- 4) Lặp lại các bước 2) và 3) cho đến khi chúng ta đạt đến vị trí mà mỗi đỉnh của đồ thị được bao gồm trong S hoặc T , tức là $S \cup T = V$. Khi đó T là một tập cực đại độc lập.

Để tìm tất cả các tập độc lập, chúng ta sẽ sử dụng đệ quy. Trong bước 2) của thuật toán, chúng ta sẽ không chọn một đỉnh tùy ý, nhưng chúng ta sẽ khảo sát điều gì sẽ xảy ra khi thêm mỗi đỉnh tự do của đồ thị, tức là không thuộc S và T . Điều này sẽ được thực hiện bởi hàm đệ quy `maxSubSet (last)`

```

maxSubSet(last) {
    if (<S hợp T = V>) {
        <In ra tập hợp độc lập tìm được>;
        return;
    }
    for (<mỗi đỉnh i | i thuộc V, i thuộc S, i thuộc T, i > last) {
        <Thêm i vào T>;
        <Thêm vào S nối mọi đỉnh với i>;
        maxSubSet(i);
        <Loại i khỏi T>;
        <Khôi phục S trước gọi hồi quy>;
    }
}

```

Trong đoạn trên, cuối cùng là đỉnh được thêm vào cuối cùng trong tập độc lập tối đa mà chúng ta đang xây dựng. Chúng ta chỉ chọn những đỉnh i mà $i > last$. Điều này là cần thiết để tránh tạo các tập hợp độc lập (chẳng hạn như $\{1, 3, 7\}$ và $\{1, 7, 3\}$, v.v.).

Sau đây là mã nguồn của chương trình. Đồ thị được biểu diễn bằng ma trận lân cận và dữ liệu đầu vào được đặt dưới dạng hằng số.

Chương trình 5.23. Tập ộc lập cực đại trong đồ thị (522maxindep.c)

```

#include <stdio.h>
/* Số đỉnh cực đại có thể */
#define MAXN 200
/* Số đỉnh trong ô thị đã cho */
const unsigned n = 8;
/* Ma trận kề của đ thị */
const char A[MAXN][MAXN] = {
    { 0, 1, 0, 0, 0, 1, 0, 1 },
    { 1, 0, 1, 0, 0, 1, 0, 0 },
    { 0, 1, 0, 1, 1, 0, 0, 0 },
    { 0, 0, 1, 0, 1, 0, 1, 0 },
    { 0, 0, 1, 1, 0, 1, 0, 0 },
    { 1, 1, 0, 0, 1, 0, 1, 1 },
    { 0, 0, 0, 1, 0, 1, 0, 0 },
    { 1, 0, 0, 0, 0, 1, 0, 0 } };

```

```
unsigned S[MAXN], T[MAXN], sN, tN;
```

```

void print(void)
{ unsigned i;
  printf(" ");
  for (i = 0; i < n; i++)
    if (T[i]) printf("%u ", i + 1);
  printf("} \n");
}

void maxSubSet(unsigned last)
{ unsigned i, j;
  if (sN + tN == n) {
    print();
    return;
  }
  for (i = last; i < n; i++) {
    if (!S[i] && !T[i]) {
      for (j = 0; j < n; j++)
        if (A[i][j] && !S[j]) {
          S[j] = last+1; sN++;
        }
      T[i] = 1; tN++;
      maxSubSet(i+1); /* Hồi quy */
      T[i] = 0; tN--;
      for (j = 0; j < n; j++)
        if (S[j] == last+1) { S[j] = 0; sN--; }
    }
  }
}

int main() {
  unsigned i;
  printf("Tất cả tập độc lập tối đa trong đồ thị là:\n");
  sN = tN = 0;
  for (i = 0; i < n; i++) S[i] = T[i] = 0;
  maxSubSet(0);
  return 0;
}

```

Kết quả thực hiện chương trình:

Tất cả các tập độc lập tối đa trong đồ thị là:

- $\{1 3 7\}$
- $\{1 4\}$
- $\{1 5 7\}$
- $\{2 4 8\}$
- $\{2 5 7 8\}$
- $\{3 6\}$
- $\{3 7 8\}$
- $\{4 6\}$

Bài tập

- ▷ 5.71. Chứng minh rằng thuật toán được đề xuất để tìm tất cả các tập độc lập tối đa hoạt động đúng đắn.
- ▷ 5.72. Đề xuất cách giải quyết bài toán trên.

5.7.3. Tập đỉnh trội

Định nghĩa 5.32. Một đồ thị có định hướng $G(V, E)$ đã cho. *Tập hợp các đỉnh trội* được gọi là tập S sao cho:

- $S \subseteq V$
- Mọi đỉnh không thuộc S đều là con trực tiếp của đỉnh nào đó từ S .

Định nghĩa 5.33. Một tập hợp đỉnh trội được gọi là *tối thiểu* nếu nó không chứa tập hợp đỉnh trội khác làm tập hợp con của chính nó.

Hãy xem một ví dụ minh họa thuật ngữ được giới thiệu.

Bài toán: Có một số dịch giả nói các ngôn ngữ khác nhau, những người này sẽ dịch từ tiếng Việt sang các ngôn ngữ tương ứng. Trong Bảng 5.4 chúng ta đã đánh dấu người dịch bằng các chữ cái A, B, C, D, E và đối với các ngôn ngữ họ nói, chúng ta đã đặt 1 vào vị trí tương ứng.

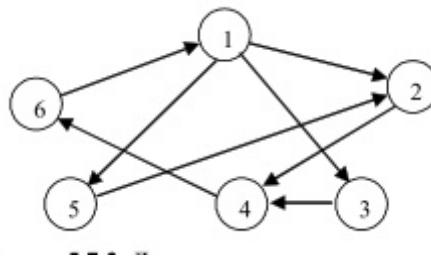
Chọn số lượng người dịch tối thiểu để chúng ta có một người dịch cho từng ngôn ngữ trong số bảy ngôn ngữ được liệt kê ở trên. Ví dụ, nếu chúng ta chọn B, C và D , điều kiện bắt buộc sẽ được đáp ứng.

	A	B	C	D	E
Tiếng Anh	1	0	1	1	0
Tiếng Pháp	1	1	0	0	0
Tiếng Đức	0	1	0	0	0
Tiếng Ý	1	0	0	1	0
Tiếng Nga	0	1	1	0	1
Tiếng Trung	0	0	0	1	1
Tiếng Phần Lan	0	0	1	0	0

Bảng 5.4. Người dịch.

Bài toán này và các bài toán khác, mà chúng ta sẽ xem xét sau này, được giải quyết bằng cách tìm các tập hợp chi phối tối thiểu trong một đồ thị (Trong ví dụ này, độ phức tạp tính toán đa thức [Manev-1996]).

Tập đỉnh trội có thể có nhiều hơn một và với một số phần tử khác nhau. Nếu chúng ta biểu thị bằng p_1, p_2, \dots, p_k số phần tử của mỗi tập đỉnh trội tối thiểu của một đồ thị đã cho, thì $p = \min p_1, p_2, \dots, p_k$ được gọi là *số trội* đối với đồ thị.



Hình 5.29. Tập đỉnh trội trong đồ thị

Cho đồ thị Hình 5.29 các tập đỉnh trội là $\{1, 4\}$, $\{1, 4, 6\}$, $\{3, 5, 6\}$, v.v., và các tập hợp $\{1, 4\}$ và $\{3, 5, 6\}$ là các tập hợp chi phối tối thiểu. $\{1, 4, 6\}$ chiếm ưu thế nhưng không tối thiểu (bản thân nó chứa $\{1, 4\}$). Số trội của đồ thị là 2.

Định nghĩa 5.34. Một đồ thị có hướng $G(V, E)$ và một tập T , $T \subseteq V$

được cho trước. Phủ của tập T được gọi là tập $P(T)$, bao gồm tất cả các đỉnh của T và tất cả các đỉnh kề với một đỉnh của T .

Thuật toán để tìm tất cả các tập đỉnh trội tối thiểu

Cho T là tập rỗng. Ở mỗi bước, chúng ta sẽ thêm một đỉnh mới vào T , sao cho sau khi bổ sung, điều kiện sau vẫn được đáp ứng:

(*) không có đỉnh $i \in T$ sao cho $P(T \setminus \{i\}) \equiv P(T)$.

Nếu ở bước nào đó $P(T) \equiv V$, thì chúng ta đã tìm thấy một tập hợp trội tối thiểu.

Để tìm tất cả các tập hợp trội tối thiểu, chúng ta sẽ xây dựng T theo tất cả các cách có thể bằng cách vết cạn hoàn toàn.

Trong cách triển khai được hiển thị bên dưới, hàm `ok()` xác minh rằng điều kiện trên được đáp ứng cho bất kỳ tập nào T . Việc xây dựng T được thực hiện theo sơ đồ đệ quy sau:

```
void findMinDom(int last) {
    if (P(T) == V) {
        <In ra tập trội tìm được>;
        return;
    }
    for (<mỗi đỉnh i của đồ thị, i >= last>) {
        T = T hợp với {i};
        if (ok()) findMinDom(i+1);
        T = T \ {i};
    }
}
```

Triển khai mã nguồn

Chương trình 5.24. Tập trội tối thiểu trong đồ thị (523mindom.c)

```
#include <stdio.h>
/* Số đỉnh cực đại có thể */
#define MAXN 200
/* Số đỉnh trong đồ thị */
const unsigned n = 6;
/* Ma trận cạnh kề */
const char A[MAXN][MAXN] = {
    { 0, 1, 1, 0, 1, 0 },
    { 0, 0, 0, 1, 0, 0 },
    { 0, 0, 0, 0, 1, 1 },
    { 1, 0, 0, 0, 0, 1 },
    { 1, 1, 0, 0, 0, 0 },
    { 0, 0, 1, 1, 0, 0 }
};
```

```

    { 0, 0, 0, 1, 0, 0 },
    { 0, 0, 0, 0, 0, 1 },
    { 0, 1, 0, 0, 0, 0 },
    { 1, 0, 0, 0, 0, 0 }
};

unsigned cover[MAXN];
char T[MAXN];

void printSet(void)
{
    unsigned i;
    printf("{ ");
    for (i = 0; i < n; i++)
        if (T[i])
            printf("%u ", i + 1);
    printf("}\n");
}

char ok(void)
{
    unsigned i, j;
    for (i = 0; i < n; i++) if (T[i]) {
        /* kiểm tra xem lớp phủ có được bảo toàn không khi bỏ đỉnh i*/
        if (0 == cover[i]) continue;
        for (j = 0; j < n; j++) if (cover[j])
            if (!(cover[j] - A[i][j]) && !T[j])
                break; /* còn đỉnh không phủ */
        if (j == n) return 0;
    }
    return 1;
}

void findMinDom(unsigned last)
{
    unsigned i, j;
    /* kiểm tra phải chăng lời giải tìm được*/
    for (i = 0; i < n; i++)
        if (!cover[i] && !T[i]) break;
    if (i == n) { printSet(); return; }
    /* không - tiếp tục xây dựng tập trội */
    for (i = last; i < n; i++) {
        T[i] = 1;
        for (j = 0; j < n; j++) if (A[i][j]) cover[j]++;
    }
}

```

```

if (ok()) findMinDom(i + 1);
for (j = 0; j < n; j++) if (A[i][j]) cover[j]--;
T[i] = 0;
}
}

int main() {
unsigned i;
printf("Các tập trội nhỏ nhất trong đồ thị là: \n");
for (i = 0; i < n; i++) { cover[i] = 0; T[i] = 0; }
findMinDom(0);
return 0;
}

```

Kết quả thực hiện chương trình:

Các tập trội tối thiểu trong đồ thị là:

{1 2 6}
{1 3 6}
{1 4}
{3 5 6}

Bài tập

Chứng minh rằng thuật toán được đề xuất để tìm tất cả các tập trội ối thiểu hoạt động chính xác.

5.7.4. Tập cơ sở

Bài toán: Một mạng máy tính N được đã cho và các kênh để liên lạc giữa chúng được biết đến. Xác định một số lượng tối thiểu của máy tính (cơ sở) sao cho khi một số thông tin cần được phổ biến, nó có thể tiếp cận tất cả các máy tính trong mạng từ cơ sở.

Trong Hình 5.30. Máy tính được biểu diễn dưới dạng các đỉnh của biểu đồ định hướng. Một cơ sở có thể trong biểu đồ là bộ {4, 7, 9}. Nó không phải là duy nhất: như vậy là {5, 7, 9}, {7, 8, 9} và những người khác.

Định nghĩa 5.35. Một đồ thị có định hướng $G(V, E)$ đã cho. Một *tập cơ sở* của các đỉnh (hoặc chỉ *tập cơ sở*) được gọi là *tập hợp* $T \subseteq V$ sao cho:

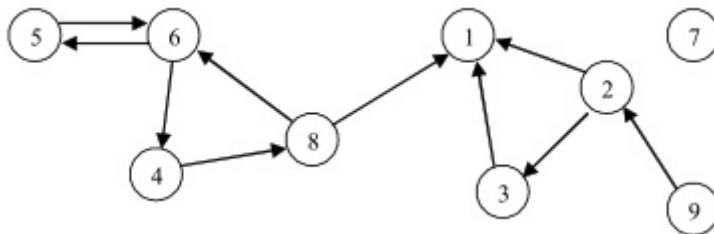
- mỗi đỉnh có thể tiếp cận được bằng một đỉnh T
- T cực tiểu theo nghĩa là không có tập con thích hợp của T tương ứng với điều kiện đầu tiên cho khả năng tiếp cận.

Mệnh đề 5.4. *Đây là cơ sở nếu và chỉ khi nó được thực hiện đồng thời:*

- 1) *Mỗi đỉnh không tham gia T có thể đạt tới một đỉnh T .*
- 2) *Không có đỉnh $i \in T$ nào có thể đạt tới đỉnh khác từ T .*

Mệnh đề 5.5. *Trong một đồ thị có chu trình có định hướng, có một cơ sở T duy nhất và nó bao gồm tất cả các đỉnh có bậc đầu vào bằng 0 (tức là chúng không có đỉnh trước nó).*

Theo định nghĩa (cũng như từ Mệnh đề 5.4 và Mệnh đề 5.5), suy ra các cơ sở trong đồ thị có thể là một số, nhưng chúng phải bằng số đỉnh.



Hình 5.30. Đồ thị có hướng

Thuật toán tìm cơ sở

Cơ sở chúng ta sẽ gọi những đỉnh tham gia vào cơ sở. Chúng ta sẽ nhập một base[], chẳng hạn như $\text{base}[i] = 1$, khi i là cơ sở và $\text{base}[i] = 0$, ngược lại. Ban đầu, chúng ta đánh dấu tất cả các đỉnh là cơ sở. Chúng ta duyệt theo chiều sâu mọi đỉnh k , với $\text{cbase}[k] = 1$. Khi đi ngang, tất cả các đỉnh i ($i \neq k$) mà chúng ta đi qua sẽ được đánh dấu là $\text{base}[i] = 0$. Do đó, các đỉnh vẫn được đánh dấu là cơ sở sẽ tạo thành một cơ sở trong đồ thị. Độ phức tạp của thuật toán được mô tả là $\Theta(n + m)$, bởi vì mỗi đỉnh và mỗi cạnh được vượt qua nhiều nhất một lần (Tại sao?). Triển khai đầy đủ như sau:

Chương trình 5.25. Các đỉnh tạo thành tập cơ sở trong đồ thị (524v-base.c)

```
#include <stdio.h>
/* Số đỉnh cực đại có thể trong đồ thị */
#define MAXN 200
/* Số đỉnh trong đồ thị */
const unsigned n = 9;
/* Ma trận liền kề của đồ thị */
const char A[MAXN][MAXN] = {
    { 0, 0, 0, 0, 0, 0, 0, 0, 0 },
    { 1, 0, 1, 0, 0, 0, 0, 0, 0 },
    { 1, 0, 0, 0, 0, 0, 0, 0, 0 },
    { 0, 0, 0, 0, 0, 0, 0, 1, 0 },
    { 0, 0, 0, 0, 0, 1, 0, 0, 0 },
    { 0, 0, 0, 1, 1, 0, 0, 0, 0 },
    { 0, 0, 0, 0, 0, 0, 0, 0, 0 },
    { 1, 0, 0, 0, 0, 1, 0, 0, 0 },
    { 0, 1, 0, 0, 0, 0, 0, 0, 0 }
};

char used[MAXN], base[MAXN];

void DFS(unsigned i)
{ unsigned k;
    used[i] = 1;
    for (k = 0; k < n; k++)
        if (A[i][k] && !used[k]) {
            base[k] = 0;
            DFS(k);
        }
}

void solve(void) {
    unsigned i, j;
    for (i = 0; i < n; i++) base[i] = 1;
    for (i = 0; i < n; i++)
        if (base[i]) {
            for (j = 0; j < n; j++) used[j] = 0;
            DFS(i);
        }
}
```

```

}

void print() {
    unsigned i, count = 0;
    printf("Các đỉnh tạo ra tập cơ sở trong đồ thị là: \n");
    for (i = 0; i < n; i++)
        if (base[i]) { printf("%u ", i + 1); count++; }
    printf("\nSố đỉnh trong tập cơ sở: %u\n", count);
}

int main() {
    solve();
    print();
    return 0;
}

```

Kết quả thực hiện chương trình:

Các đỉnh tạo thành cơ sở trong đồ thị là:

4 7 9

Số lượng đỉnh trong cơ sở: 3

Bài tập

- ▷ 5.73. Chứng minh Mệnh đề 5.4 và Mệnh đề 5.5).
- ▷ 5.74. Hãy chứng minh rằng thuật toán được đề xuất để tìm tập cơ sở dữ liệu hoạt động đúng.

5.7.5. Tâm, bán kính và đường kính

Các bài toán để chọn một hoặc một số đỉnh cố định của đồ thị sao cho đáp ứng một tiêu chí tối ưu nhất định được sử dụng rộng rãi. Ví dụ, chúng ta hãy xem các đỉnh của một đồ thị là các vùng lân cận và giải thích các cạnh liên thông là các đường dẫn trực tiếp giữa chúng. Chúng ta đang tìm kiếm một đỉnh từ đồ thị để phục vụ như một dịch vụ khẩn cấp (hoặc cứu hỏa, cảnh sát, tổng đài điện thoại, v.v.) sao cho khoảng cách đến vùng lân cận xa nhất là nhỏ nhất. Bài toán này được gọi là *tối ưu hóa tình huống xấu nhất*.

Cũng có thể tìm kiếm tập hợp các đỉnh p (một số dịch vụ khẩn

cấp), trong trường hợp đó, mục đích sẽ là giảm thiểu khoảng cách xa nhất $S(i)$ từ đỉnh i đến dịch vụ khẩn cấp gần nhất.

Định nghĩa 5.36. Một đồ thị có hướng liên thông $G(V, E)$ đã cho. Số lượng phân tách bên ngoài $S_o(i)$ cho một đỉnh $i \in V$ cho trước được gọi là $d(i, j)$ lớn nhất, đối với mỗi $j \in V$, trong đó $d(i, j)$ là độ dài của đường đi nhỏ nhất từ i đến j . Nghĩa là, số này là đường đi ngắn nhất đến đỉnh xa nhất của i . Tương tự, số lần phân tách trong $S_t(i)$ cho đỉnh i được gọi là giá trị lớn nhất của khoảng cách nhỏ nhất từ một số đỉnh $j \in V$ đến đỉnh i .

Định nghĩa 5.37. Từ tất cả các đỉnh, chúng ta sẽ chọn một đỉnh $i \in V$ mà $p_o = S_o(i)$ là cực tiểu. Đỉnh i được gọi là *tâm ngoài* của đồ thị G , và số p_o được gọi là bán kính ngoài của đồ thị. Tương tự, chúng ta sẽ chọn một đỉnh j sao cho $p_t = S_t(j)$ là cực tiểu. Đỉnh j này được gọi là *tâm bên trong* của G , và p_t được gọi là bán kính bên trong của đồ thị.

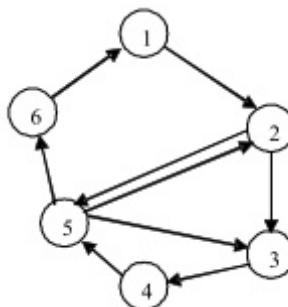
Định nghĩa 5.38. Gọi $S_{ot}(i)$ biểu thị giá trị nhỏ nhất của $d(i, j) + d(j, i)$, được tính cho mỗi $j \in V$. Gọi v là đỉnh mà $S_{ot}(v)$ là cực tiểu. Khi đó v được gọi là *tâm ngoài-trong* (hoặc chỉ *tâm*), và giá trị tương ứng $S_{ot}(v)$ được gọi là bán kính ngoài-trong (hoặc chỉ bán kính) của đồ thị.

Lưu ý rằng trong cả ba định nghĩa, đỉnh được đề cập có thể không phải là định nghĩa duy nhất.

Ví dụ về một trung tâm trong đồ thị: Chúng ta muốn đặt một đòn cản sát sao cho nếu S là khoảng cách nhỏ nhất từ nó đến bất kỳ vùng lân cận nào cộng với khoảng cách nhỏ nhất trên đường về thì S đến vùng lân cận xa nhất sẽ là nhỏ nhất.

Trước khi tổng quát bài toán (p -tâm và p -bán kính trong đồ thị), chúng ta sẽ chỉ ra tâm trong đồ thị như thế nào. Hãy xem xét một ví dụ cụ thể (xem Hình 5.31).

Chúng ta đã tìm thấy các đường đi ngắn nhất trong đồ thị (được hiển thị bên phải trong Hình 5.31) theo thuật toán Floyd. Sau bước cơ bản này, tâm và bán kính được xác định bằng cách thực hiện trực tiếp định nghĩa ở trên. Độ phức tạp của thuật toán là $\Theta(n^3)$. (Tại sao?)



Hình 5.31. Tâm và bán kính trong đồ thị (các cạnh có trọng số 1)

	1	2	3	4	5	6
1	0	1	2	3	2	3
2	3	0	1	2	1	2
3	4	3	0	1	2	3
4	3	2	2	0	1	2
5	2	1	1	2	0	1
6	1	2	3	4	3	0

Bảng 5.5. Tâm của đồ thị: đỉnh 2

Trong cách triển khai bên dưới, đồ thị định hướng được biểu diễn bằng ma trận trọng số $A[][]$, vì số đỉnh của nó là n . Dữ liệu đầu vào được đặt ở đầu chương trình dưới dạng hằng số.

Chương trình 5.26. Tâm và bán kính của đồ thị (525a-center.c)

```
#include <stdio.h>
/* Số đỉnh lớn nhất có thêr */
#define MAXN 150
#define MAX_VALUE 10000
/* Số đỉnh đồ thị đã cho */
const unsigned n = 6;
/* Ma trận trọng số*/
int A[MAXN][MAXN] = {
    { 0, 1, 0, 0, 0, 0 },
    { 0, 0, 1, 0, 1, 0 },
    { 0, 0, 0, 1, 0, 0 },
    { 0, 1, 2, 3, 4, 3 },
    { 2, 3, 0, 1, 2, 1 },
    { 1, 2, 3, 4, 3, 0 }
};
```

```

{ 0, 0, 0, 0, 1, 0 },
{ 0, 1, 0, 0, 0, 1 },
{ 1, 0, 0, 0, 0, 0 }
};

/* Tìm độ dài của đường ngắn nhất giữa mọi cặp điểm*/
void floyd(void)
{
    unsigned i, j, k;
    /*Các giá trị 0 thay đổi của MAX_VALUE */
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if (A[i][j] == 0)
                A[i][j] = MAX_VALUE;

    /* Thuật toán floyd*/
    for (k = 0; k < n; k++)
        for (i = 0; i < n; i++)
            for (j = 0; j < n; j++)
                if (A[i][j] > (A[i][k] + A[k][j]))
                    A[i][j] = A[i][k] + A[k][j];
    for (i = 0; i < n; i++)
        A[i][i] = 0;
}

void findCenter(void)
{
    unsigned i, j, center;
    int max, min;
    min = MAX_VALUE;
    /* Sot(Xi) = max {Vj [d(Xi, Xj)+d[Xj,Xi]] } , tâm là đỉnh X* */
    /* với nó Sot(X*) là nhỏ nhất */
    for (i = 0; i < n; i++) {
        max = A[i][0] + A[0][i];
        for (j = 0; j < n; j++)
            if ((i != j) && ((A[i][j] + A[j][i]) > max))
                max = (A[i][j] + A[j][i]);
        if (max < min) {
            min = max; center = i;
        }
    }
    printf("Tâm của đồ thị là đỉnh %u\n", center + 1);
}

```

```

printf("Bán kính của đồ thị là %u\n", min);
}

int main() {
    floyd();
    findCenter();
    return 0;
}

```

Kết quả thực hiện chương trình:

Tâm của đồ thị là đỉnh 2

Bán kính của đồ thị là 4

Bài tập

- ▷ 5.75. Cho một ví dụ về một đồ thị với một số tâm có thể có.
- ▷ 5.76. Hãy sửa đổi chương trình trên để nó tìm thấy tất cả các tâm có thể có của đồ thị.

- *p-tâm* và *p-bán kính*

Tổng quát bài toán trên là chọn đồng thời p số đỉnh tâm của đồ thị. Chúng ta muốn giảm thiểu khoảng cách xa nhất từ bất kỳ đỉnh nào và đến đỉnh gần nhất trong số các đỉnh đã chọn. Gọi $d(i, j)$ là độ dài của đường đi nhỏ nhất giữa các đỉnh i và j .

Định nghĩa 5.39. Đồ thị định hướng $G(V, E)$ và tập $Q = \{v_{i1}, v_{i2}, \dots, v_{ip} | v_{ik} \in V\}$. Chúng ta đưa vào ký hiệu:

$$S(v, Q) = \min\{d(v_k, v) + d(v, v_k) | v_k \in Q\}, v \in V$$

$$S(Q) = \max\{S(v, Q) | v \in V\}$$

$S(Q)$ nhỏ nhất, được tính trên tất cả các tập con p phần tử có thể có Q của V , được gọi là *p-tâm* của đồ thị.

Đây là định nghĩa trong trường hợp đơn giản hơn, khi biểu đồ không có hướng:

Định nghĩa 5.40. Đồ thị vô hướng $G(V, E)$ và tập $Q = \{v_{i1}, v_{i2}, \dots, v_{ip} | v_{ik} \in V\}$. Chúng ta đưa vào ký hiệu:

$$S(v, Q) = \min\{d(v_k, v) | v_k \in Q\}, v \in V$$

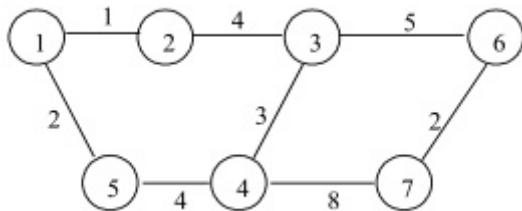
$$S(Q) = \max S(v, Q) | v \in V$$

$S(Q)$ nhỏ nhất, được tính trên tất cả các tập con p phần tử có thể có Q của V , được gọi là p -tâm của đồ thị.

Thuật toán tìm p -tâm

Chúng ta sẽ xem xét trường hợp đồ thị vô hướng. Nếu đồ thị được định hướng, những thay đổi là không đáng kể. Giải pháp bao gồm việc khai thác hoàn toàn các biến thể có thể có: với mỗi tập con p -phân tử Q , $Q \subseteq V$, chúng ta sẽ tính khoảng cách S đến đỉnh xa nhất. Trong tất cả các khả năng (tổng $\binom{p}{n}$ theo số), chúng ta sẽ giữ nguyên những khả năng cho giá trị nhỏ nhất của S .

Ban đầu, chúng ta tính toán độ dài của các đường đi nhỏ nhất giữa mỗi hai đỉnh bằng cách sử dụng thuật toán Floyd. Khi chúng ta tìm thấy ma trận có các đường đi nhỏ nhất $A[][],$ chúng ta sẽ tạo ra tất cả các tổ hợp có thể có (tất cả các tập con p -phân tử có thể có Q của V) và với mỗi tổ hợp, chúng ta sẽ tính khoảng cách từ nó đến đỉnh xa nhất. Điều này không khó lầm, vì chúng ta có ma trận Floyd theo ý của nó. Ví dụ cụ thể như Hình 5.32.



Hình 5.32. Tìm p -tâm

Chúng ta hãy tìm kiếm một 3-tâm, tức là $p = 3$ trong đồ thị Hình 5.32. Chúng ta đã tìm thấy ma trận Floyd và xem xét tập con của các đỉnh $\{2, 3, 6\}$:

$$\begin{aligned} S &= \max\{\min\{A[i][2], A[i][3], A[i][6]\} | i = 1, 2, \dots, 7\} \\ &= \max\{1, 0, 0, 3, 3, 0, 2\} = 3. \end{aligned}$$

Hơn nữa, kiểm tra tất cả các khả năng khác, chúng ta sẽ thấy rằng $\{2, 3, 6\}$ là tập tối thiểu hóa S và theo đó nó là 3-tâm bắt buộc

	1	2	3	4	5	6	7
1	0	1	5	6	2	10	12
2	1	0	4	7	3	9	11
3	5	4	0	3	7	5	7
4	6	7	3	0	4	8	8
5	2	3	7	4	0	12	12
6	10	9	5	8	12	0	2
7	12	11	7	8	12	2	0

Bảng 5.6. Ma trận trọng số của đồ thị

trong đồ thị. Tuy nhiên, nó không phải là duy nhất! Ví dụ, đối với tập $\{1, 3, 6\}$ chúng ta lại nhận được 3-bán kính $S = 3$.

Điều tiếp theo là sự nhận ra của C, mà chỉ tìm thấy một giải pháp. Chúng ta đề nghị người đọc sửa đổi nó theo cách thích hợp để ta tìm ra tất cả các giải pháp.

Chương trình 5.27. Tìm p -tâm trong đồ thị trong đồ thị (524526p-center.c)

```
#include <stdio.h>
/* Số đỉnh lớn nhất có thể trong ô thi */
#define MAXN 150
#define MAX_VALUE 10000
/* Số đỉnh đồ thị đã cho */
const unsigned n = 7;
const unsigned p = 3; /* p-tâm */
/* Ma trận trọng số của đồ thị */
int A[MAXN][MAXN] = {
    { 0, 1, 0, 0, 2, 0, 0 },
    { 1, 0, 4, 0, 0, 0, 0 },
    { 0, 4, 0, 3, 0, 5, 0 },
    { 0, 0, 3, 0, 4, 0, 8 },
    { 2, 0, 0, 4, 0, 0, 0 },
    { 0, 0, 5, 0, 0, 0, 2 },
    { 0, 0, 0, 8, 0, 2, 0 }
};
```

```

unsigned center[MAXN], pCenter[MAXN], pRadius;
/* Tìm độ dài của đường ngắn nhất giữa hai cặp đỉnh */
void floyd(void)
{ unsigned i, j, k;

/* Các giá trị 0 thay đổi của MAX_VALUE */
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        if (A[i][j] == 0) A[i][j] = MAX_VALUE;

/* Thuật toán floyd */
for (k = 0; k < n; k++)
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if (A[i][j] > (A[i][k] + A[k][j]))
                A[i][j] = A[i][k] + A[k][j];
    for (i = 0; i < n; i++)
        A[i][i] = 0;
}

/* Tính S cho tập con vừa sinh ra */
void checkSol(void)
{ unsigned i, j, cRadius = 0;
    int bT = MAX_VALUE;
    for (i = 0; i < p; i++)
        if (A[center[i]][j] < bT) bT = A[center[i]][j];
        if (cRadius < bT) cRadius = bT;
    }
    if (cRadius < pRadius) {
        pRadius = cRadius;
        for (i = 0; i < p; i++) pCenter[i] = center[i];
    }
}

/* Tổ hợp C(n,p) - sinh ra tất cả tập con p-phần tử của G */
void generate(unsigned k, unsigned last)
{ unsigned i;
    for (i = last; i < n - p + k; i++) {
        center[k] = i;
}

```

```

if (k == p)
    checkSol();
else
    generate(k + 1, i + 1);
}

/*
In ra p-tâm và p-bán kính*/
void printPCenter(void)
{
    unsigned i;
    printf("%u-Tâm của đồ thị là tập hợp các đỉnh sau: {", p);
    for (i = 0; i < p; i++) printf("%d ", pCenter[i] + 1);
    printf("}\n");
    printf("%u-bán kính của ô thị là %u\n", p, pRadius);
}

int main()
{
    floyd();
    pRadius = MAX_VALUE;
    generate(0, 0);
    printPCenter();
    return 0;
}

```

Kết quả thực hiện chương trình:

- 3-Tâm của đồ thị là tập hợp các đỉnh sau: {1 3 6}
3-Bán kính của đồ thị là 3

Bài tập

- ▷ 5.77. Kiểm tra xem tập {2,3,6} có phải là p -tâm của đồ thị trong Hình 5.32 hay không.
- ▷ 5.78. Cho một ví dụ về một đồ thị với một số p -tâm có thể có.
- ▷ 5.79. Sửa đổi chương trình trên để nó tìm thấy tất cả các p -tâm có thể.
- ▷ 5.80. Thuật toán trên có hoạt động trong trường hợp đồ thị có định hướng không? Nếu có thì có thay đổi nào được yêu cầu không (nếu có: cái gì?), Nếu không - tại sao (đưa ra một ví dụ cụ thể)?

5.7.6. Kết hợp cặp. Kết hợp cặp tối đa

Việc tìm kiếm kết hợp theo từng cặp tối đa trong một đồ thị (đồi sánh) được liên kết với một số bài toán của các luồng trong biểu đồ mà chúng ta đã xem xét.

Định nghĩa 5.41. Kết hợp cặp trong đồ thị $G(V, E)$ được gọi là tập con $E' \subseteq E$ sao cho, trong đó mỗi đỉnh của đồ thị là điểm cuối của không quá một cạnh của E' . Các đỉnh trùng với một cạnh của E' được gọi là phủ, và những đỉnh không phải là đầu của các cạnh của kết hợp kép được gọi là tự do. Tương tự, chúng ta xác định các *cạnh bao phủ* và *tự do* tùy thuộc vào việc chúng có thuộc kết hợp kép hay không.

Định nghĩa 5.42. Một tổ hợp ghép đôi E' từ k cạnh được gọi là *cực đại* nếu với mọi kết hợp ghép đôi khác E'' của l cạnh $k \geq l$ được thỏa mãn.

Định nghĩa 5.43. Gọi $M \subseteq E$ là một kết hợp kép tùy ý trong G . Một đường đi trong G được gọi là *xen kẽ đối với* M nếu một trong hai cạnh liên tiếp của đường đi, đúng một đường tham gia vào M . Một đường đi xen kẽ với kết hợp M được gọi là *tăng* nếu đỉnh ban đầu và đỉnh cuối cùng của nó là tự do đối với M .

Thuật toán tìm kết hợp cặp tối đa

Chúng ta bắt đầu với một kết hợp kép ngẫu nhiên (có thể bao gồm một cạnh duy nhất). Ở mỗi bước, chúng ta tìm thấy một con đường tăng dần liên quan đến cặp kết hợp cuối cùng. Nếu một con đường như vậy không tồn tại, nó là tối đa. (Tại sao?) Nếu không, chúng ta xây dựng một kết hợp cặp mới có nhiều cạnh hơn tổ hợp đầu tiên như sau: Gọi M là kết hợp cặp và P là đường tăng. Sau đó, kết hợp theo từng cặp mới sẽ chứa tất cả các cạnh của M và P không thuộc cả M và P .

Chúng ta cung cấp cho người đọc một bài toán thực sự phức tạp như bài tập: vẽ một sơ đồ để tìm một đường đi tăng dần cần thiết cho việc thực hiện các thuật toán.

Trong trường hợp đồ thị hai phần, mọi thứ được đơn giản hóa rất nhiều:

Định nghĩa 5.44. Đồ thị vô hướng $G(V, E)$ được gọi là lưỡng phân nếu có một phân hoạch $V = V' \cup V'', V' \cap V'' = \emptyset$, sao cho mỗi cạnh $(i, j) \in E$ thỏa mãn $i \in V'$ và $j \in V''$.

Bài toán: Tìm kết hợp kép lớn nhất trong đồ thị hai phần.

Thuật toán 1

Chúng ta sẽ rút gọn bài toán thành bài toán tìm dòng cực đại trong đồ thị.

Cho đồ thị hai phần $G(V, E)$ bằng cách phá vỡ tập các đỉnh $V = V' \cup V'', V' \cap V'' = \emptyset$. Chúng ta sẽ giải quyết bài toán bằng cách tìm dòng chảy lớn nhất trong đồ thị mà:

- các đỉnh của tập V' là nguồn.
- các đỉnh của tập V'' là người tiêu dùng.
- tất cả các cạnh của đồ thị đều có độ thông qua $c(i, j) = 1$.

Thuật toán 2

Thuật toán tổng quát (được mô tả ở trên) để tìm kết hợp nhị phân tối đa trong một đồ thị tùy ý có thể dễ dàng thực hiện trong trường hợp đồ thị lưỡng phân. Trong trường hợp này, số lượng đường đi tăng lên bị giới hạn bởi $\Theta(n)$ và bạn có thể dễ dàng tìm thấy một đường đi đang tăng lên bằng cách thu thập thông tin theo chiều rộng (Làm thế nào?). Như vậy tổng độ phức tạp là $\Theta(n.(M + 1)) = \Theta(n.m)$. Ngoài ra còn có các thuật toán hiệu quả hơn để tìm kết hợp nhị phân tối đa trong đồ thị hai phần, với độ phức tạp $\Theta(\sqrt{n}.m)$ [Hopcroft-Karp-1973], sau này được tóm tắt cho một đồ thị tùy ý [Micali, Vazirani-1980].

Bài tập

- ▷ 5.81. Hãy chứng minh rằng nếu không có đường tăng nào của một kết hợp đã cho trong đồ thị thì kết hợp đó là cực đại.
- ▷ 5.82. Hãy vẽ sơ đồ tìm đường đi tăng dần, cần thiết cho việc thực hiện thuật toán tổng quát để tìm kết hợp tối đa.
- ▷ 5.83. Hãy vẽ sơ đồ tìm đường đi tăng dần trong đường hai phần với duyệt theo chiều rộng.
- ▷ 5.84. Hãy thực hiện triển khai thuật toán 1 và 2 ở trên.

5.8. Tô màu và đồ thị phẳng

5.8.1. Tô màu đồ thị và sắc số

Nhiều bài toán, chẳng hạn như lập kế hoạch quy trình sản xuất, lập lịch trình, lưu trữ và vận chuyển hàng hóa, có thể dễ dàng giải quyết nếu chúng được giảm xuống thành bài toán tô màu đồ thị tương ứng.

Định nghĩa 5.45. r -tô màu các đỉnh của đồ thị được gọi là so khớp chính xác một phần tử của một tập hợp $\{c_1, c_2, \dots, c_r\}$ đã cho trên mỗi đỉnh của nó. Màu r của các cạnh của đồ thị được xác định tương tự.

Định nghĩa 5.46. Một đồ thị vô hướng $G(V, E)$ được gọi là r -sắc số nếu các đỉnh của nó có thể có màu r sao cho mọi hai đỉnh liền kề có màu khác nhau. Số r nhỏ nhất mà đồ thị có màu r được gọi là *số sắc đỉnh* của G (hoặc chỉ số màu của G) và được ký hiệu là $\gamma_V(G)$. Tương tự, số r nhỏ nhất sao cho các cạnh của đồ thị có thể tô màu r để không có các cạnh ngẫu nhiên được tô cùng màu được gọi là *số sắc cạnh* của G và được ký hiệu là $\gamma_E(G)$.

Sau đây là hai ví dụ:

Ví dụ 1: Chúng ta cần sắp xếp các báo cáo của đại hội sao cho không người tham gia nào bị buộc phải bỏ sót một báo cáo mà họ muốn tham dự. Thời lượng tối thiểu của chương trình là bao nhiêu nếu chúng ta có đủ số lượng giảng đường để có thể gửi bất kỳ báo cáo nào cùng một lúc? Trong ví dụ này, chúng ta có thể xây dựng một đồ thị G , các đỉnh của chúng là các báo cáo. Hai đỉnh sẽ được kết nối với nhau khi và chỉ khi có một người tham gia muốn đến thăm cả hai người họ. Giải pháp cho vấn đề này là tìm $\gamma_V(G)$, cung cấp thời lượng tối thiểu của chương trình.

Ví dụ 2: Mỗi người trong số n doanh nhân muốn có cuộc gặp trực tiếp với một số người khác. Chúng ta đang tìm thời gian tối thiểu mà các cuộc họp sẽ diễn ra, nếu mỗi cuộc họp kéo dài một ngày và có chính xác hai người tham gia. Vấn đề có thể được giải quyết bằng cách tìm số sắc độ của đồ thị $\gamma_E(G)$, cho chúng ta số ngày cần thiết.

- Giới hạn dưới cùng của số sắc

Trong phần tiếp theo, chúng ta sẽ xem xét các thuật toán để tìm số sắc độ của một đồ thị. Sau đó, chúng ta sẽ thấy rằng một ràng buộc có thể có ở dưới cùng của $\gamma_V(G)$ theo một số tiêu chí sẽ tiết kiệm rất nhiều thời gian tính toán. Các tiêu chí như vậy đã cho các tuyênl bố sau (một số trong số đó tuân theo rõ ràng, trong khi tính hợp lệ của các tiêu chí khác được kiểm tra ít tần thường hơn):

- $\gamma_V(G) \geq 2 \Leftrightarrow G$ chứa ít nhất một cạnh.
- $\gamma_V(G) \geq 3 \Leftrightarrow G$ chứa một vòng lặp có độ dài lẻ.
- $\gamma_E(G) \geq d_m(G)$, với $d_m(G)$ là tung độ cực đại của đỉnh G .
- $\gamma_V(G) \geq n/\gamma_V(G')$, với n là số đỉnh của G và G' là phần bù của G .
- $\gamma_V(G) \geq n^2/(n^2 - 2m)$, với n là số đỉnh và m là số cạnh của G .

- Tìm sắc số đỉnh

Thuật toán tìm $\gamma_V(G)$

1) Sử dụng các phát biểu đã thảo luận trong đoạn trước, chúng ta giới hạn số màu $\gamma_V(G)$ bên dưới bằng một số $\gamma_{V_{\min}}$.

```
2) for (r = gamma_{Vmin}; r <n; r++) {
if (G là r-sắc số) {Chuyển đến bước 3}
3) r là sắc số của đồ thị.
```

Rõ ràng, bài toán là kiểm tra xem một đồ thi có phải là r -chromatic hay không. Vấn đề thứ hai là một bài toán NP-đầy đủ và chúng ta sẽ giải quyết nó trong ??: Kiểm tra tuần tự tất cả các màu có thể có của mỗi đỉnh, không tiếp tục tô màu, nếu đã có hai đỉnh liền kề được tô cùng màu. Cách tiếp cận toàn diện này có hiệu suất tốt đối với đồ thi có số lượng đỉnh nhỏ, đối với đồ thi thừa thớt và đồ thi gần như hoàn chỉnh. Có các thuật toán toàn diện khác để tìm số sắc độ đỉnh. Ví dụ, có thể tìm thấy các tập độc lập lớn nhất của đồ thi: các đỉnh thuộc một tập độc lập có thể được tô cùng màu. Đối với đồ thi ngẫu nhiên, đây là một trong những thuật toán tốt nhất [Christofides-1975] [Korman-1979].

Nói chung, để đảm bảo chúng ta tìm đúng số màu chính xác,

chúng ta phải áp dụng một thuật toán dựa trên sự cạn kiệt hoàn toàn. Tuy nhiên, trong thực tế, chúng ta thường không quan tâm đến chính xác màu r tối thiểu (số màu thực), mà quan tâm đến một số giá trị gần đúng đủ tốt của nó. Đôi với những trường hợp như vậy, có những thuật toán giải quyết vấn đề một cách nhanh chóng, nhưng không phải lúc nào cũng chính xác. Những điều này sẽ được thảo luận lại trong Chương 9.

Bài tập

- ▷ 5.85. Cố gắng chứng minh các tiêu chí ràng buộc của $\gamma_V(G)$.

5.8.2. Đồ thị phẳng

Định nghĩa 5.47. Một đồ thị được gọi là *đồ thị phẳng*, vì vậy nó có thể được vẽ trên một mặt phẳng (vẽ bằng đồ thị với các dấu chấm và đường thẳng) sao cho không có hai cạnh nào cắt nhau.

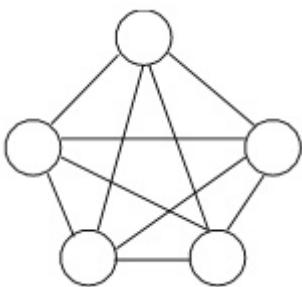
Ví dụ, một đồ thị mô tả các hành lang của một phòng trưng bày là phẳng. Bản đồ chính trị của thế giới cũng có thể được biểu diễn bằng một biểu đồ phẳng (các đỉnh tương ứng với các quốc gia và mỗi cạnh thể hiện một đường biên giới trên bộ chung giữa hai quốc gia).

Biết rằng bất kỳ đồ thị phẳng nào cũng có thể được vẽ trong mặt phẳng sao cho không những không có hai cạnh nào cắt nhau mà còn có các cạnh là *đoạn thẳng*.

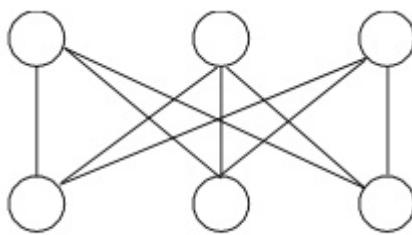
Năm 1933 Kuratovski lần đầu tiên xác định chặt chẽ khái niệm độ phẳng của một đồ thị và đưa ra điều kiện cần và đủ cho đồ thị phẳng.

Chúng ta sẽ nhắc lại Định nghĩa 5.8. và chúng ta sẽ cung cấp thêm một vài định nghĩa cần thiết cho định lý phẳng cơ bản của đồ thị:

Cho các đồ thị vô hướng $G(V, E)$ và $G'(V, E')$ và với mỗi $i, j \in V$ thì cạnh (i, j) thuộc đúng một trong hai tập E' và E . Khi đó G' được gọi là *phản bù* của G . Nếu $(i, j) \in E$ thỏa mãn với mỗi $i, j \in V$ thì đồ thị được gọi là *đầy đủ*. Chúng ta sẽ biểu thị một đồ thị đầy đủ với n đỉnh bằng K_n (K_5 được thể hiện trong Hình 5.33).



Hình 5.33. \$K_5\$



Hình 5.34. \$K_{3,3}\$

Định nghĩa 5.48. Nếu trong đồ thị hai phần \$G(V, E)\$ với phân hoạch \$V = V' \cup V''\$, \$V' \cap V'' = \emptyset\$, với mỗi \$i \in V'\$ và \$j \in V''\$ thì suy ra \$(i, j) \in E\$, thì chúng ta nói rằng \$G(V = V' \cap V'', E)\$ là một đồ thị hai phần đầy đủ.

Nếu số phần tử của \$V'\$ là \$m\$ và số phần tử của \$V''\$ là \$n\$ thì đồ thị có kí hiệu là \$K_{m,n}\$. Trong Hình 5.34 được hiển thị \$K_{3,3}\$.

Định nghĩa 5.49. Sự co lại của một đồ thị \$G(V, E)\$ được gọi là đồ thị \$G'(V', E')\$, nhận được từ \$G\$ bằng cách loại bỏ tất cả các đỉnh bậc 2 và thay thế hai cạnh qua mỗi đỉnh như vậy bằng một cạnh.

Định nghĩa 5.50. Hai đồ thị \$G\$ và \$G'\$ được gọi là *đẳng cấu* (viết \$G' \approx G''\$) nếu tồn tại biểu diễn ánh xạ lên của các đỉnh của \$G'\$ trong các đỉnh của \$G''\$ bảo toàn lân cận.

Định lý 5.9 (Kuratowski). Một đồ thị là phẳng bởi vì không có đồ thị con nào của nó, mà sự co lại của nó là đẳng cấu thành \$K_5\$ hoặc \$K_{3,3}\$.

Định lý 5.10 (cho bốn màu). Sắc số của đồ thị phẳng nhỏ hơn hoặc bằng 4.

Định lý cuối cùng có nghĩa là chúng ta luôn có thể tô màu một bản đồ chính trị với bốn màu để không có các quốc gia láng giềng cùng màu. Nó được trình bày như một giả thuyết vào năm 1850. Nhiều nỗ lực không thành công sau đó đã được thực hiện để chứng minh điều đó, mãi đến năm 1976 K. Appel và W. Hacken mới thành công. Bằng chứng bao gồm phân tích 2.000 trường hợp khác nhau

và được thực hiện với hơn 1.000 giờ sử dụng máy tính. Ngày nay, có rất nhiều bằng chứng về nó, hầu hết trong số đó lại sử dụng các thuật toán và tính toán máy tính.

Ngoài ra còn có các thuật toán xác minh đa thức [Chiba, Nishizeki, Abe, Ozawa-1985], và trong trường hợp đồ thị phẳng, chúng cũng tìm biểu diễn phẳng tương ứng. Tuy nhiên, những vấn đề này bao gồm tài liệu đủ rộng và sẽ không được xem xét ở đây.

Bài tập

▷ 5.86. Định lý nghịch đảo cho bốn màu có đúng không, tức là nếu sắc số của một đồ thị nhỏ hơn 5, thì nó có phải là đồ thị phẳng không?

5.9. Câu hỏi và bài tập

Trong phần cuối cùng, một số bài toán lý thuyết thú vị được đính kèm, cũng như các bài toán được đã cho tại các cuộc thi lập trình dành cho học sinh và trường học ở Hungary và quốc tế.

▷ 5.87. *Bài toán cho Người đưa thư Trung Quốc, Duke programming contest, 1992, problem f.*

Bài toán này (từ Bài toán Người đưa thư Trung Quốc) được đặt theo tên của nhà toán học Trung Quốc Mei Guo Guang).

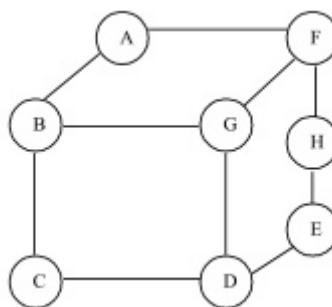
Tìm một vòng có độ dài nhỏ nhất trong một đồ thị có trọng số đi qua mỗi cạnh ít nhất một lần. Đồ thị không nhất thiết phải là Euler (Nếu là Euler thì chu trình Euler bất kỳ thỏa mãn điều kiện).

▷ 5.88. *Duke programming contest, 1993, problem g.* Danh sách các quan hệ giữa các biến kiểu $i < j$ được đã cho. Viết chương trình in ra tất cả các cách sắp xếp có thể có của các biến tương thích với các quan hệ đã cho. Ví dụ, đối với các quan hệ $x < y$ và $x < z$, có hai cách sắp xếp: xyz và xzy .

▷ 5.89. *Duke programming contest, 1993, problem h.* Các hình vuông được kết nối bởi các đường một chiều được đã cho. Viết chương trình tìm số đường nằm giữa hai hình vuông đã cho. Có thể có vô số con đường giữa hai hình vuông (ví dụ, nếu bạn đi qua một vòng). Trong trường hợp này, chương trình phải in -1.

► 5.90. ACM Regional contest, New Zealand, 1991, problem A. Một đồ thị vô hướng $G(V, E)$ được đã cho. Nếu một phép thứ tự được cho trên các đỉnh của nó, thì B -bậc của đỉnh $v \in V$ được xác định như khoảng cách dài nhất giữa v và mọi đỉnh lân cận.

B -Bậc của đồ thị được gọi là cực đại của các B -bậc riêng lẻ cho mỗi đỉnh.



Hình 5.35. B -bậc của đồ thị

Ví dụ, đối với đồ thị của Hình 5.35 và đối với hai thứ tự mẫu (A, B, C, D, E, H, F, G) và (A, B, C, D, G, F, H, E) B -bậc của các đỉnh là: 6, 6, 1, 4, 1, 1, 6, 6 cho thứ tự đầu tiên và 5, 3, 1, 4, 3, 5, 1, 4 - cho thứ tự thứ hai, nó cung cấp B -bậc cho đồ thị là 6 ở thứ nhất, và 5 - trong trường hợp thứ hai.

Viết chương trình tìm thứ tự của các đỉnh mà B -bậc của đồ thị là cực tiểu.

► 5.91. chu trình đơn tối thiểu.

Một đồ thị có trọng số không định hướng được đã cho. Tìm độ dài tối thiểu của một vòng lặp đơn giản trong một cột có ít nhất 3 đỉnh.

Lời giải: Thực tế là một đồ thị đã cho không cần thiết trong trường hợp này. Nếu có nhiều hơn một cạnh giữa hai đỉnh i và j , thì chúng ta chỉ để lại giá trị nhỏ nhất của chúng (trong một chu trình đơn, mỗi đỉnh có thể tham gia nhiều nhất một lần - vì vậy không thể có nhiều hơn một cạnh giữa hai đỉnh tham gia vào nó. chỉ cạnh nhỏ nhất, vì chúng ta muốn tìm chu trình nhỏ nhất). Sau đây, giải như sau:

Thuật toán 1. Với mỗi cạnh (i, j) của đồ thị ta thực hiện:

- chúng ta loại trừ (i, j) khỏi đồ thị;
- chúng ta tìm đường đi nhỏ nhất giữa i và j . Một chu trình có chiều dài L_{ij} thu được cùng với cạnh bị loại trừ.

L_{ij} tối thiểu là độ dài của chu trình tối thiểu cần thiết.

Độ phức tạp của thuật toán là $\Theta(m.n.\log 2n)$.

Thuật toán 2. Cho:

- $d(i, j)$ là độ dài của đường đi nhỏ nhất giữa các đỉnh i và j .
- $f(i, j)$ là trọng lượng của cạnh (i, j) .

Khi đó độ dài của chu trình đơn nhỏ nhất là giá trị nhỏ nhất của tổng $d(i, k) + f(k, j) + d(j, i)$, với mỗi i, j và k mà đường đi nhỏ nhất giữa i và k không có điểm chung với đường đi nhỏ nhất giữa j và i .

Giải thích cách thu được độ phức tạp được chỉ ra của *thuật toán 1*. Tính độ phức tạp của *thuật toán 2*.

▷ 5.92. chu trình nhỏ nhất qua đỉnh k

Đã cho một số tự nhiên k cố định. Đề xuất một thuật toán cho một đa đồ thị có trọng số không định hướng tùy ý $G(V, E)$ tìm thấy một chu trình đơn có độ dài nhỏ nhất trong G chứa ít nhất k đỉnh.

Rõ ràng, $k = n$ dẫn đến bài toán Người bán hàng, tuy nhiên bài toán này không là NP -đầy đủ (xem Chương 6), vì độ phức tạp của nó được xác định bởi n và k là cố định.

Độ phức tạp tốt nhất là đa thức n mà bạn có thể đạt được để giải quyết bài toán, với điều kiện k là hằng số tùy ý nhưng cố định?

▷ 5.93. Cơ sở số trong đồ thị

Định nghĩa 5.51. Cho trước một đồ thị có hướng $G(V, E)$ và một hàm $v : V \rightarrow \mathbb{R}^+$, tập các trọng số của các đỉnh của đồ thị. Css của các đỉnh được gọi là tập $T \subseteq V$ sao cho:

- mỗi đỉnh của V có thể truy cập được bởi một số đỉnh của T
- tổng các trọng số của các đỉnh của T là nhỏ nhất.

Tìm cơ sở số của các đỉnh của đồ thị.

Có hai vấn đề thoát nhìn không liên quan gì đến đồ thị, nhưng trong thực tế chúng ta dễ dàng giải quyết được nếu chúng ta trình bày chúng bằng đồ thị và tìm đường đi dài nhất trong đó (xem ??).

▷ 5.94. *Số lượng tối đa không giảm*

Tìm dãy số không giảm lớn nhất của một dãy số đã cho. Hay nói đúng hơn: Cho một dãy số nguyên a_1, a_2, \dots, a_n . Tìm một dãy a_{i1}, a_{i2}, \dots , và a_{ik} ($i1 < i2 < \dots < ik$) sao cho $a_{ij} \leq a_{i(j+1)}$ với $j = 1, 2, \dots, k - 1$, và tổng $S = \sum_{i=1,2,\dots,k} a_{ij}$ là cực đại.

▷ 5.95. *Chèn ba chiều*

Có n hộp được xác định bởi tọa độ các đỉnh của chúng (x_i, y_i, z_i) . Tìm số hộp tối đa để chúng có thể xếp thành một hàng.

Gợi ý: Trong bài toán này, chúng ta có thể xây dựng một đồ thị trong đó các đỉnh sẽ đại diện cho các hộp và các cạnh được định hướng sẽ cho biết liệu một hộp có thể vừa với hộp khác hay không.

▷ 5.96. *Đường dẫn nhỏ nhất trong đồ thi chu trình*

Có thể cải thiện độ phức tạp của thuật toán Dijkstra nếu chúng ta tìm kiếm một đường đi nhỏ nhất chỉ giữa hai đỉnh trong một đồ thị không?

▷ 5.97. *Góc ngược*

Một đồ thị có định hướng được đã cho. Hãy lập một thuật toán để kiểm tra xem có một đỉnh trong đồ thị với một bậc vào ở đầu vào $n - 1$ và một bậc đầu ra là 0.

▷ 5.98. *Kiểm tra xem biểu đồ có phải là cặp đôi không*

Một đồ thị có n đỉnh và m cạnh đã cho. Kiểm tra với độ phức tạp $\Theta(m + n)$ xem đồ thị đã cho có phải là cặp đôi hay không.

Gợi ý: Sử dụng một sửa đổi của chức năng duyệt theo chiều sâu.

▷ 5.99. *Kiểm tra tính đẳng cấu của đồ thi*

Độ phức tạp tốt nhất mà bạn có thể đạt được để xác minh rằng hai đồ thị đã cho là đẳng cấu?

▷ 5.100. *Bổ sung một đồ thi phẳng*

Cho trước một đồ thi phẳng $G(V, E)$ với n ($n \geq 11$) đỉnh. Kiểm tra phần bù của G là một đồ thi phẳng.

Gợi ý: Để chứng tỏ rằng phần bù của G không thể là một đồ thi phẳng, bất kể G thuộc loại nào.

► 5.101. *Tạp chí máy tính*-5/1994

Một đồ thị vô hướng được đã cho. Đối với một số đỉnh trong đồ thị, một số thực được cho trước - trọng số của đỉnh. Kiểm tra xem có thể so sánh các số thực trên các đỉnh khác hay không, sao cho trọng số của mỗi đỉnh bằng trung bình cộng của trọng số các đỉnh lân cận của nó.

Gợi ý: Biểu đồ được chia thành các thành phần kết nối. Tất cả các đỉnh trong một thành phần phải có cùng trọng số để đáp ứng điều kiện của nhiệm vụ.

► 5.102. *d-nén*

Cho trước một ma trận vuông $A[] []$ với kích thước $n \times n$, bao gồm các số không và đơn vị. Đã biết số lượng đơn vị trong mỗi hàng và mỗi cột của ma trận. Hãy khôi phục ma trận.

► 5.103. *Olympic lập trình Balkan*

Hệ thống dầu là một bể chứa n ($2 \leq n \leq 200$) có thể tích từ 1 đến 100 và $n - 1$ ống nối các bể trong một hệ thống kín (không có chu trình). Hệ thống được lắp đầy bởi một trong các bồn chứa S (nguồn), bồn này luôn trống và được sử dụng như một máy bơm. Trong một đơn vị thời gian, một đơn vị dầu đi qua mỗi ống nối với nguồn S . Quá trình tiếp tục cho đến khi tất cả các bể chứa đầy.

Đối với hệ thống bể chứa được kết nối, hãy tìm nguồn S tối ưu mà từ đó quá trình làm đầy yêu cầu tối thiểu thời gian.

Lời giải: Một đồ thị vô hướng xoay chiều liên thông $G(V, E)$ với các trọng số đỉnh được cho. Sau khi loại bỏ nguồn S , đồ thị được chia thành k ($2 \leq k \leq n - 1$) thành phần kết nối. Gọi tổng trọng số của các đỉnh trong thành phần thứ i của kết nối là T_i . Khi đó, nguồn S phải được chọn sao cho $T = \max\{T_i\}$, với $i = 1, 2, \dots, k$ là cực tiểu.

Độ phức tạp tốt nhất có thể đạt được để giải quyết vấn đề sao cho bộ nhớ được sử dụng là $\Theta(n)$?

► 5.104. *Tách các đỉnh (các cạnh)*

Một đồ thị vô hướng được cho trong đó hai đỉnh s và t được chọn. Tìm số phần tử của tập đỉnh (cạnh) H nhỏ nhất của đồ thị sao cho nếu các đỉnh (cạnh) tham gia H bị loại khỏi đồ thị thì s và t vẫn nằm trong các thành phần liên thông khác nhau.

▷ 5.105. Dòng chảy trong một đồ thị với tính thấm của các cạnh của số thực

Bài toán tìm lưu lượng cực đại trong đồ thị sẽ thay đổi như thế nào nếu trọng số đã cho (độ cho qua được) của các cạnh là số thực?

Gợi ý: Độ phức tạp của thuật toán trong trường hợp này sẽ phụ thuộc vào trọng số của các cạnh và kích thước của luồng. Một thuật toán có độ phức tạp chỉ phụ thuộc vào số đỉnh n của đồ thị không được biết đến.

▷ 5.106. Bổ sung cho một đồ thị được liên thông chặt

Một đồ thị được liên thông yếu có định hướng được đã cho. Thêm số lượng cạnh tối thiểu vào đồ thị để nó trở nên liên thông chặt.

▷ 5.107. 2^n

Một số nguyên n được cho trước. Tìm một chuỗi có độ dài nhỏ nhất sao cho các biểu diễn nhị phân của tất cả các số từ 1 đến n tham gia như các tập con.

Gợi ý: Vấn đề được giải quyết bằng cách tìm chu trình Euler. Ngoài ra còn có một số thuật toán đơn giản hơn, tính đúng đắn của nó được chứng minh bằng cách xem xét các chu trình Euler.

▷ 5.108. Đường kính của đồ thị

Viết chương trình tìm đường kính của đồ thị.

Bạn có thể soạn một thuật toán có độ phức tạp nhỏ hơn $\Theta(n^3)$ không?

▷ 5.109. Phương pháp sóng

Ma trận vuông với các phần tử 0 và X và tọa độ của hai ô từ nó được cho: ban đầu (s_x, s_y) và cuối cùng (t_x, t_y) . Để tìm cách giữa chúng, tức là, một chuỗi các ô sao cho:

- Đường dẫn chỉ nên bao gồm các ô có giá trị 0.
- Mỗi hai ô liên tiếp trong đường dẫn phải là "liền kề", tức là khác nhau chính xác một trong hai tọa độ của chúng.
- Tổng chiều dài của con đường phải nhỏ nhất.

Giải pháp: Vấn đề được giải quyết một cách hiệu quả với cái gọi là phương pháp sóng (diễn giải cụ thể của kỹ thuật thu thập thông tin theo chiều rộng). Phương pháp này bao gồm những điều sau:

1) Tại vị trí ban đầu ta viết số 1. Ta đặt $i = 1$.

2) Chúng ta tìm tất cả các ô trong đó số i được viết. Trong tất cả các hàng xóm tự do của chúng (tức là các ô có giá trị 0), chúng ta viết số $i + 1$.

3) Nếu một trong các lân cận từ bước 2 trùng với vị trí cuối cùng, thuật toán kết thúc (tìm được đường đi nhỏ nhất và độ dài của nó là $i + 1$). Nếu không, chúng ta tăng i và lặp lại bước 2).

X	X	X	X	X	X	X	X
X	1	2	3	4	5	6	X
X	2	X	X	X	6	7	X
X	3	4	5	X	8	X	X
X	4	X	6	7	8	X	X
X	5	6	7	X	X	X	X
X	6	7	8	9	X	X	X
X	X	X	X	X	X	X	X

Hình 5.36. Phương pháp sóng.

Phương pháp này được gọi là phương pháp sóng, bởi vì việc tìm kiếm đường đi với tiến trình nhất quán theo mọi hướng có thể tương tự như cách nước được phân phối trong một không gian kín, với vị trí bắt đầu của nguồn. Hiệu quả của việc thực hiện thuật toán trên được xác định bởi cách tìm kiếm các lảng giềng trong mỗi bước.

CHƯƠNG 6

THUẬT TOÁN QUAY LUI

6.1. Phân loại bài toán	489
6.1.1. Phức tạp về thời gian	489
6.1.2. Độ phức tạp tính toán theo bộ nhớ	490
6.1.3. Bài toán không thể giải được	490
6.1.4. Các ví dụ	490
6.2. Bài toán NP-đầy đủ	495
6.3. Tìm kiếm với quay lui	498
6.3.1. Sự thỏa mãn của một hàm Boolean	500
6.3.2. Tô màu đồ thị	507
6.3.3. Đường đi đơn dài nhất trong đồ thị chu trình ..	511
6.3.4. Đường đi quân ngựa	514
6.3.5. Bài toán tám quân Hậu	519
6.3.6. Thời khóa biểu của trường học	524
6.3.7. Dịch mật mã	529
6.4. Phương pháp nhánh và ranh giới	534
6.4.1. Bài toán ba lô (lựa chọn tối ưu)	535
6.5. Các chiến lược tối ưu cho trò chơi	539
6.5.1. Trò chơi "X" và "O"	541
6.5.2. Nguyên tắc minimum và maximum	546
6.5.3. Nhát cắt alpha-beta	548
6.5.4. Duyệt alpha-beta đến một độ sâu nhất định ..	551
6.6. Câu hỏi và bài tập	553

Người ta thường chấp nhận rằng một vấn đề được coi là "được giải quyết tốt" nếu độ phức tạp của thuật toán tương ứng là tuyến tính hoặc đa thức (mức độ thấp). Tuy nhiên, chúng ta thường gặp các tác vụ quan trọng mà không có thuật toán nào "nhanh". Các bài toán này sẽ là chủ đề của chương này, và các kỹ thuật khác nhau để giải quyết chúng sẽ không chỉ được thảo luận ở đây mà còn ở các chương sau.

6.1. Phân loại bài toán

Khi chúng ta nói về các lớp bài toán, chúng ta hiểu việc nhóm các bài toán theo mức độ phức tạp của chúng - các bài toán có độ phức tạp tương tự được xếp vào cùng một lớp. Có nhiều tiêu chí khác nhau để phân loại bài toán theo độ "khó" của chúng. Chúng ta sẽ xem xét việc phân loại bài toán trên hai tiêu chí đánh giá quan trọng nhất: thời gian và bộ nhớ.

6.1.1. Phức tạp về thời gian

Hiện tại, chúng ta sẽ giới hạn bản thân chỉ xem xét các bài toán có thể xác minh (tức là những bài toán yêu cầu câu trả lời có hoặc không).

Bài toán NP

NP (từ thời gian đa thức không xác định) là lớp của các *bài toán có thể tính toán thời gian đa thức*. Một bài toán thuộc về lớp NP, nếu với độ phức tạp tính toán đa thức có thể kiểm tra xem một ứng viên có thực sự là một giải pháp hay không, mà không cần quan tâm sẽ mất bao lâu để tìm thấy một ứng viên như vậy. Nói một cách đơn, bài toán NP là những bài toán mà khi chúng ta cố gắng "tìm" ra ứng viên phù hợp cho một giải pháp, chúng ta có thể dễ dàng kiểm tra nó (xem ví dụ 2 của 6.1.4).

Bài toán P

Đây là những bài toán về khả năng tính toán mà có một giải pháp với *độ phức tạp của đa thức* (chữ viết tắt *P* bắt nguồn từ từ *polynom* trong tiếng Anh). Ngược lại, lớp này (là một lớp con của lớp NP) khá rộng: khi xem xét các bài toán thực tế, bậc của hàm đa thức là vô cùng quan trọng. Lớp này cũng chứa các thuật toán mà hàm của nó có tiệm cận tăng thấp hơn hàm tuyến tính. Đó là hàm logarit (thường các tác vụ này được xem xét trong một lớp con riêng biệt - Bài toán log), hàm Ackermann nghịch đảo và các hàm khác. (xem ví dụ 1 của 6.1.4).

Bài toán lũy thừa

Đây là những bài toán có thể được giải quyết với độ phức tạp theo lũy thừa. Lớp này chứa hai phần trước, nhưng không toàn diện

- có những bài toán mà các thuật toán tốt nhất có độ phức tạp cao hơn lũy thừa.

6.1.2. Độ phức tạp tính toán theo bộ nhớ

Trong cách phân loại này, bộ nhớ được sử dụng như một chức năng có kích thước của dữ liệu đầu vào là rất quan trọng. Chúng ta không quan tâm đến độ phức tạp của thuật toán để giải quyết chúng. Lớp này có thể được chia thành nhiều lớp con: *P-space* (các tác vụ yêu cầu bộ nhớ đa thức), *exp-space* (bộ nhớ hàm mũ) và các lớp khác. (xem ví dụ 3 của 6.1.4).

Rõ ràng là độ phức tạp tính toán của thuật toán luôn ít nhất bằng độ phức tạp của bộ nhớ. Thật vậy, không thể chiếm bộ nhớ có kích thước $\Theta(n)$ trong thời gian nhỏ hơn $\Theta(n)$ [Manev-1996].

6.1.3. Bài toán không thể giải được

Có những bài toán thuật toán mà [Manev-1996] có thể chứng minh rằng chúng không thể giải được, bất kể chúng ta có bao nhiêu thời gian và bộ nhớ. Tiếp theo chúng ta sẽ đưa ra các ví dụ cụ thể (xem ví dụ 4 của 6.1.4).

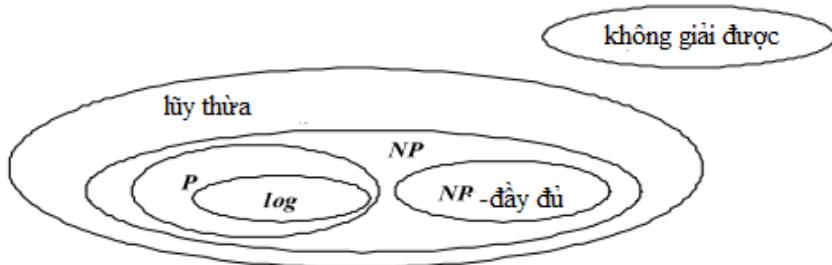
6.1.4. Các ví dụ

Một số lớp bài toán được liệt kê (và các kết nối giữa chúng) được thể hiện dưới dạng giản đồ trong Hình 6.1. Hình này cũng cho thấy loại bài toán *NP*-đầy đủ, chúng ta sẽ thảo luận chi tiết hơn trong 6.2. Hình 6.1 cho thấy một ví dụ về vị trí của các bài toán, vì vị trí chính xác của chúng không được xác định. Một trong những câu hỏi lớn trong tin học lý thuyết là liệu $P \subseteq NP$ hay $P \equiv NP$. Giả thuyết đầu tiên và Hình 6.1 được coi là có nhiều khả năng hơn, phản ánh điều này.

Chúng ta sẽ xem xét một số ví dụ về các bài toán thuộc các lớp khác nhau.

Ví dụ 1. Các bài toán có thể giải bằng đa thức

Một số ví dụ về các bài toán đa thức có thể giải được có thể được liệt kê, nhiều trong số đó đã được thảo luận trong bài báo cho đến nay: các thuật toán từ lý thuyết đồ thị (để duyệt đồ thị và các sửa



Hình 6.1. Phân loại bài toán (theo thời gian).

đổi của chúng, thuật toán Dijkstra, Floyd, Prim, Kruskal và nhiều thuật toán khác - tất nhiên, chúng ta phải xác định chúng là các bài toán có thể xác minh), để tìm kiếm (ví dụ, tìm kiếm nhị phân - với độ phức tạp $\Theta(\log_2 n)$), hầu hết tất cả các thuật toán sắp xếp, để tính toán các hàm toán học khác nhau và nhiều hàm khác.

Ví dụ 2. Bài toán NP

Ví dụ đơn giản sau đây minh họa rõ ràng ý nghĩa của lớp các bài toán *NP*: Một số tự nhiên n là cho trước. Kiểm tra xem có ước của số tự nhiên n nhỏ hơn số tự nhiên q ($1 < q < n$) hay không.

Nếu chúng ta có một ứng cử viên cho nghiệm q' , chúng ta có thể dễ dàng kiểm tra xem có p' sao cho $n = p'.q'$ hay không, nhưng đối với lời giải của bài toán trên thì tài nguyên tính toán cần thiết còn nhiều hơn nữa.

Hãy xem xét một ví dụ khác: một đồ thị $G(V, E)$, hai đỉnh $i, j \in V$ và một số tự nhiên k được cho trước.

Bài toán: Kiểm tra xem có một đường đi đơn (không có đỉnh lặp lại) giữa i và j với độ dài ít nhất k .

Bài toán này thuộc về lớp NP: nếu chúng ta đã có được một đường dẫn, chúng ta có thể dễ dàng (với độ phức tạp tuyến tính) kiểm tra điều kiện xem đường đi có đơn không và nó có dài ít nhất k . Tuy nhiên, một thuật toán có độ phức tạp đa thức để tìm đường như vậy vẫn chưa được biết đến, vì vậy chúng ta không thể chắc chắn liệu bài toán có thuộc về lớp *P* hay không.

Chúng ta sẽ lưu ý sau rằng bài toán này thuộc về một lớp bài toán mới được đề cập trong phần giới thiệu và sẽ là chủ đề chính

của chương này - lớp bài toán đầy đủ NP. Đối với họ, người ta cho rằng không có nghiệm đa thức.

Ví dụ 3. Phân loại theo bộ nhớ

Trong hầu hết các loại hình thể thao "trí tuệ" (ví dụ như cờ vua, các trò chơi bài khác nhau, v.v.), việc tham gia vào máy tính đã được tổ chức từ lâu. Ban đầu, trong những lần va chạm đầu tiên như vậy, các "tay đua" máy tính hầu như không có cơ hội, và chiến thắng của con người trước cỗ máy đã hỗ trợ cho luận điểm rằng trí tuệ tự nhiên khó có thể bị các quá trình tính toán nhân tạo cạnh tranh. Tuy nhiên, nhiều người ủng hộ quan điểm này đã "uống một cốc nước lạnh" khi vào năm 1997, Deep Blue, một máy tính IBM, lần đầu tiên đánh bại nhà vô địch cờ vua thế giới Gary Kasparov trong một trận đấu chính thức. Năm 1996, Deep Blue thắng một ván trước Kasparov, nhưng thua trận (tổng cộng 6 ván). Tuy nhiên, sau đó, vào năm 1997, anh đã đánh bại anh ta với 3,5: 2,5 điểm (2 thắng, 1 thua và 3 hòa), nhờ đó anh nhận được giải thưởng 100.000 đô la mà anh đã thiết lập 17 năm trước đó cho chương trình máy tính đầu tiên của mình. nhà vô địch cờ vua thế giới. Tuy nhiên, chúng ta sẽ lưu ý rằng khả năng của máy tính hiện đại vẫn không cho phép một chương trình máy tính trở thành nhà vô địch thế giới trong cờ vua, và nếu có thể, điều này vẫn chưa được chứng minh trong thực tế: Deep Blue chỉ thắng một phần trong các ván cờ., và sau khi anh ấy được lập trình đặc biệt để thi đấu tốt với Kasparov. [Russell, Norvig-1995] [Lovicchio-1997]

Tuy nhiên, ở những trò chơi đơn giản hơn như cờ caro, cỗ máy đã chứng tỏ được ưu thế của mình so với con người. Năm 1992, chương trình máy tính Chinook, được tạo ra bởi Jonathan Schaefer và các đồng nghiệp của ông, đã giành chức vô địch Mỹ mở rộng, nhưng không thể trở thành nhà vô địch thế giới, do Marion Tinsley giành chiến thắng với tỷ số 21,5: 18,5. Tinsley đã 40 năm vô địch thế giới (!) Và chỉ để thua 3 trận chính thức trong thời gian đó. Tuy nhiên, trong cuộc đụng độ với Chinook, anh đã ghi được trận thua thứ tư và thứ năm. Sau đó, vào năm 1994, Chinook trở thành nhà vô địch thế giới sau khi trận đấu chính thức gặp Tinsley bị đình chỉ vì lý do sức khỏe.

Nhưng hãy quay lại với cờ vua. Lập trình các chương trình cờ vua là gì? Chuỗi nước đi có thể được hiểu là một cái cây: trước mặt

người chơi bắt đầu trò chơi, 20 bước di chuyển khác nhau có thể được thực hiện, với mỗi bước di chuyển có nhiều bước di chuyển ứng phó khác nhau, v.v. Các chương trình cờ vua đi xung quanh cây này và kiểm tra hậu quả có thể xảy ra khi thực hiện mỗi nước đi có thể xảy ra.

Bài toán đối với một chương trình chơi cờ vua hoàn hảo là của lớp *exp – space*, nếu chúng ta coi kích thước của bàn cờ là kích thước của dữ liệu đầu vào. Ý nghĩa của việc phân loại trí nhớ? Trong cờ vua, kích thước của bàn cờ là không đổi - 8×8 . Do đó, việc duy trì tất cả các cấu hình có thể có và nước đi tốt nhất cho mỗi người trong số họ yêu cầu bộ nhớ liên tục, mặc dù hằng số này rất lớn (giả sử rằng bất kỳ cấu hình cờ vua nào cũng có thể thực hiện được thì số cấu hình cờ vua khác nhau là 65^{32} - tại sao?). Mặt khác, chiều cao của cây cờ không lớn: một ván cờ dài hơn 150 nước đi rất hiếm khi được chơi. Do đó, nếu chúng ta tìm kiếm trên cây cho mỗi bước đi, vẫn đề thiêu bộ nhớ sẽ biến mất, nhưng một vấn đề khác lại nảy sinh: thời gian dài (theo cấp số nhân) không thể chấp nhận được để tính toán mỗi nước đi.

Có những phương pháp thu thập thông tin được gọi là cắt ngắn alpha-beta (xem 6.5.3) hạn chế việc nghiên cứu cây quyết định, nhưng thậm chí chúng không thể xử lý số lượng lớn các đỉnh thu thập thông tin. Vì lý do này, các chương trình cờ vua thực sự tìm kiếm ít chuyên sâu hơn (ví dụ: 8 quân tiến lên) và từ thông tin thu thập được, cũng như dựa trên một số tiêu chí "chiến thuật" (ví dụ: số lượng và quân cờ còn lại trên bảng, v.v.). xem 6.5.4) xác định nước đi tốt nhất có thể.

Ví dụ 4. Các bài toán có thể giải được

Bài toán sau là không thể giải được: Hai chuỗi x và y và một tập hợp các quy tắc thay thế v được đưa ra. Kiểm tra xem chuỗi x có thể được chuyển đổi thành chuỗi y hay không bằng một loạt các ứng dụng quy tắc thay thế. Các quy tắc thay thế được áp dụng như sau: mỗi khi chuỗi con v tham gia vào chuỗi tạm thời (phép biến đổi cuối cùng thu được), nó có thể được thay thế bằng u (đối với một quy tắc $v \rightarrow u$ cho trước).

Hầu hết các bài toán liên quan đến hành vi của một chương trình cũng không thể giải quyết được. Cuối cùng, chúng ta sẽ xem xét

ví dụ sau: Khi một chương trình chạy trong một thời gian dài mà không cho kết quả, rất khó để đánh giá liệu nó, trong ngôn ngữ "lập trình", có bị "lặp" hay đơn giản là thuật toán cần thêm thời gian để đầy đủ. . Thật vậy, sẽ vô cùng tiện lợi nếu có một phương pháp tự động để kiểm tra như vậy (ví dụ, trình biên dịch có thể tự đánh giá). Tuy nhiên, bài toán này là không thể giải quyết được: không có thuật toán nào có thể luôn và không có lỗi xác định chương trình có nằm trong vòng lặp vô hạn hay không. Sau đây là một ví dụ về những gì đã được nói cho đến nay:

Phần chính của chương trình

```
int main() {
    unsigned a, b, c, i, x;
    for (x = 3;;) {
        for (a = 1; a <= x; a++)
            for (b = 1; b <= x; b++)
                for (c = 1; c <= x; c++)
                    for (i = 3; i <= x; i++)
                        if (pow(a, i) + pow(b, i) == pow(c, i))
                            exit(0);
        x++;
    }
    return 0;
}
```

Đoạn được hiển thị kiểm tra xem có nghiệm của phương trình $a^n + b^n = c^n$, cho a, b, c, n số tự nhiên và $n > 2$. Theo Định lý lớn được chứng minh gần đây của Fermat, bộ số tự nhiên không tồn tại. . Đối với một người (và một người biên dịch) không biết rằng định lý đã được chứng minh, sẽ rất khó để đánh giá liệu chương trình trên có tìm ra lời giải và kết thúc hay không. Hơn nữa, có nhiều bài toán tương tự mà kết quả lý thuyết tương ứng không được biết và hoàn toàn không biết liệu các chương trình tương ứng có lặp lại hay không.

Bài toán xác định xem một vòng lặp chương trình có được gọi là vẫn đề dừng hay không. Việc xem xét lý thuyết của nó đối với một số hình thức tính toán hiện có (ví dụ, máy Turing - xem [Manev-1996]) cho phép chứng minh thực tế này một cách chặt chẽ.

Ví dụ 5. Bài toán không xác định

Đối với một số bài toán, người ta vẫn chưa xác định được liệu chúng có thuộc loại bài toán khó giải quyết hay không: một ví dụ tương tự là bài toán Scolem [MathWorld]:

Một ma trận $M_{3 \times 3}$ đã cho. Kiểm tra xem có tồn tại số tự nhiên n sao cho nếu M được nâng lên lũy thừa n thì thu được ma trận có góc trên bên phải chứa 0.

Bài tập

- ▷ 6.1. Xác định số lượng các cấu hình cờ vua khác nhau.
- ▷ 6.2. Đưa ra các ví dụ về các bài toán từ lý thuyết đồ thị có độ phức tạp lôgarit, đa thức hoặc hàm mũ.
- ▷ 6.3. Thực hiện hai bài toán sau thuộc cùng một lớp:
 - Kiểm tra số tự nhiên n có phải là số nguyên tố hay không.
 - Kiểm tra xem có ước của số tự nhiên n nhỏ hơn số tự nhiên q ($q > 1, q < n$) hay không.

6.2. Bài toán NP-đầy đủ

Từ quan điểm lý thuyết, một trong những câu hỏi thú vị nhất là liệu lớp P có trùng với NP hay không. Nói cách khác, nếu luôn có thể kiểm tra một ứng cử viên cho một nghiệm có độ phức tạp đa thức, thì liệu có thể tìm được toàn bộ lời giải với độ phức tạp như vậy không?

Người ta mong rằng câu trả lời cho câu hỏi trên là không. Tuy nhiên, vẫn chưa có bằng chứng nào chứng minh cho luận điểm này. Nghĩa là, chúng ta không thể chắc chắn rằng một vấn đề NP nhất định không nằm trong P . Loại bài toán mà chúng ta gọi là NP -đầy đủ mang lại sự rõ ràng hơn khi tìm kiếm giải pháp cho vấn đề này. NP -đầy đủ là một loại bài toán NP có thể được giảm bớt cho nhau với độ phức tạp đa thức (chúng ta sẽ xác định điều này một cách chặt chẽ trong giây lát) và mỗi bài toán từ NP có thể được giảm đa thức thành một số bài toán NP -đầy đủ. Nếu chỉ một NP -đầy đủ được chứng minh là thuộc hoặc không thuộc cấp P , thì điều này sẽ áp dụng cho tất cả các bài toán khác của cấp NP (Lưu ý từ NP -hoàn

chỉnh của cả lớp, và không chỉ từ NP -đầy đủ.). Việc khám phá ra một thuật toán đa thức cho một bài toán NP -đầy đủ là một bản tóm tắt "ý tưởng cố định" của mọi nhà toán học mới vào nghề - tìm ra một công thức đơn giản để tạo ra các số nguyên tố liên tiếp (điểm).

Định nghĩa 6.1. Chúng ta nói rằng bài toán A có thể rút gọn thành bài toán B và viết $A \langle B$ nếu có thể tìm được thuật toán giải bài A bằng cách sử dụng một số đa thức các lệnh gọi đến chương trình giải B và tất cả các phép tính bên ngoài các lệnh gọi này là độ phức tạp đa thức.

Sau đó, nếu một bài toán B thuộc loại P và $A \langle B$, thì nó theo sau rằng A cũng thuộc loại P . Để minh họa những gì đã được nói cho đến nay bằng một ví dụ: tìm một chu trình Hamilton trong một đồ thị (xem ??) - một chu kỳ trong đó mỗi đỉnh tham gia đúng một lần. Vấn đề có thể được giải quyết với sự trợ giúp của ví dụ 2:

```

for (<mỗi cạnh (i,j) từ đồ thị>) {
    if (<tồn tại đường đơn với độ dài n-1 đỉnh từ i đến j>) {
        return 1; /* Có chu trình Hamilton: đường đã tìm + cạnh (i,j)*/
    }
}
return 0;

```

Hãy biểu thị số cạnh trong đồ thị bằng m . Thuật toán sẽ tạo ra m tham chiếu đến bài toán tìm đường đi dài nhất trong đồ thị và độ phức tạp bên ngoài bài toán này sẽ là $\Theta(m)$. Nếu chúng ta áp dụng định nghĩa trên, nó có nghĩa là "Chu trình Hamilton" ' \langle '. Đường đi đơn dài nhất". Tuy nhiên, vì không có thuật toán đa thức nào được biết đến để giải "con đường đơn giản dài nhất", chúng ta không thể nói rằng chu trình Hamilton là một bài toán đa thức có thể giải được.

Định nghĩa 6.2. Một bài toán NP -A được gọi là NP -đầy đủ khi và chỉ khi đối với mọi bài toán B khác của lớp NP , nó tuân theo $B \langle A$.

Định lý 6.1 (Cook). Có ít nhất một bài toán thuộc lớp NP -đầy đủ.

Việc chứng minh định lý cuối cùng được thực hiện bằng cách tìm một bài toán NP -đầy đủ như vậy [Manev-1996] - ví dụ bài toán thỏa mãn một hàm Boolean (xem 6.3.1).

Việc áp dụng định nghĩa trên và định lý Cook cho nhiều bài toán đã dẫn đến việc chứng minh rằng chúng là *NP*-đầy đủ. Đoạn cuối cùng của chương (xem 6.6) Đưa ra các điều kiện cho hơn 70 bài toán đầy đủ *NP*. Có hàng trăm bài toán khác thuộc về lớp này. Nếu chỉ một trong số chúng chứng minh được rằng nó có thể hoặc không thể giải được bằng thuật toán đa thức, thì điều này sẽ xảy ra với tất cả những bài khác: không chỉ đối với những bài từ *NP*-đầy đủ, mà còn với *NP* nói chung. Và từ đây nó sẽ trực tiếp theo $P \equiv NP$ đó.

Trong lý thuyết về độ phức tạp và khả năng tính toán của thuật toán, các lớp mà chúng ta đã xem xét cho đến nay (P, NP, NP -đầy đủ) là các bài toán về khả năng xác minh, tức là trả lời một câu hỏi có hoặc không. Sau đây là một số ví dụ (bao gồm cả những ví dụ đã được thảo luận):

1. Để kiểm tra xem một số đêm có phải là Hamilton hay không.
2. Kiểm tra xem có chu trình Hamilton trong đồ thị có độ dài nhỏ hơn k hay không.

Tuy nhiên, có một số tác vụ không phải là tác vụ có thể xác minh được, ví dụ:

3. Tìm chu trình Hamilton có độ dài nhỏ nhất.

Trong những trường hợp này, lý thuyết cần được mở rộng và một lớp bài toán khác được đưa vào: *NP*-khó.

Định nghĩa 6.3. Bài toán A (không nhất thiết phải là bài toán kiểm chứng) thuộc cấp *NP*-khó, nếu và chỉ khi đối với mọi bài toán B thuộc cấp *NP*, nó tuân theo $B \langle A$.

Theo hệ quả của định nghĩa, dễ dàng chứng minh rằng lớp của các bài toán *NP*-đầy đủ là một phần cắt ngang của các lớp *NP* và *NP*-khó. Do đó, mối quan hệ giữa bài toán *NP*-đầy đủ và *NP*-khó có thể được biểu thị như: *NP*-đầy đủ là những bài toán *NP*-khó là những bài toán có thể xác minh được.

Đối với mỗi bài toán đầy đủ *NP*, có thể xác định một *NP*-khó tương ứng “không dễ hơn”. Ví dụ, không có giải pháp nào khác cho lời giải của Bài toán 2 ở trên (trong trường hợp tổng quát của một đồ thị tùy ý) ngoài việc tìm chu trình Hamilton nhỏ nhất (chính xác là Bài toán 3).

Bài tập

- ▷ 6.4. Chỉ sử dụng định nghĩa 6.2. để chứng minh rằng nếu một bài toán A NP -đầy đủ có thể rút gọn thành một bài toán B khác, thì B cũng là một bài toán NP -đầy đủ.
- ▷ 6.5. Đưa ra các ví dụ về các bài toán từ lý thuyết đồ thị thuộc về các bài toán NP -đầy đủ bằng cách rút gọn thành tìm kiếm chu trình Hamilton trong đồ thị (chúng ta giả định rằng tính đầy đủ NP của bài toán tìm chu trình Hamilton đã được chứng minh).
- ▷ 6.6. bài toán "Kiểm tra xem có ước nào của số tự nhiên n nhỏ hơn số tự nhiên q ($q > 1, q < n$)" thuộc lớp NP không. Và nó có thuộc lớp NP -đầy đủ không?

6.3. Tìm kiếm với quay lui

Nếu chúng ta chấp nhận giả thuyết $P \neq NP$ là đúng (ký hiệu chính xác hơn sẽ là $P \subset NP$, vì rõ ràng là $P \subseteq NP$, nhưng $P \neq NP$ được sử dụng thường xuyên hơn trong tài liệu), thì cách tiếp cận duy nhất đảm bảo rằng chúng ta sẽ luôn tìm giải pháp chính xác cho một bài toán NP -đầy đủ (hoặc NP -khó) là vét cạn hoàn toàn. Một cách có thể để đạt được mức cạn kiệt hoàn toàn là phương pháp tìm kiếm quay lui.

Phương pháp quay lui là một kỹ thuật trong đó giải pháp của một vấn đề được xây dựng một cách tuần tự. Ở mỗi bước, một nỗ lực được thực hiện để mở rộng giải pháp từng phần (không đầy đủ) hiện tại với tất cả các phần mở rộng có thể có. Nếu không có phần mở rộng nào trong số này dẫn đến một giải pháp hoàn chỉnh sau đó, trường hợp được tuyên bố là vô vọng và thuật toán quay lại một bước. Các trường hợp giới hạn cho thuật toán xảy ra khi một giải pháp hoàn chỉnh cho vấn đề được tìm thấy hoặc khi không thể mở rộng giải pháp từng phần theo bất kỳ cách nào. Lược đồ đệ quy sau đây thường được sử dụng khi giải quyết các tác vụ tìm kiếm quay lui:

Lược đồ quay lui

```
void thu(<bước i>)
{ if (i > n) <kiểm tra xem có phải là nghiệm không>;
```

```

else
/* Mở rộng nghiệm riêng theo tất cả khả năng có được*/
for (k = 1; k <= n; k++)
if (<thành phần thứ k có thể chấp nhận>) {
    <Ghi nhận là thành phần chấp nhận>;
    thu(i+1);
    <Bỏ thành phần đã ghi nhận>;
}
}

```

Trong các chương trước, phương pháp này đã được áp dụng nhiều lần. Ví dụ, trong bài toán cổ điển tìm chu trình Hamilton trong đồ thị, chúng ta sử dụng thuật toán sau:

1) Chúng ta bắt đầu xây dựng một chu trình Hamilton từ một đỉnh tùy ý $s \in V$ và đánh dấu nó như đã xét.

2) Tại mỗi bước, chúng ta cố gắng chuyển đến đỉnh v mới chưa được khám phá, sự cố của đỉnh hiện tại. Bằng cách thực hiện một quá trình chuyển đổi như vậy, chúng ta tiến thêm một bước - chúng ta đánh dấu đỉnh cao mới như đã xem xét và tiếp tục theo cách tương tự. Nếu không có đỉnh nào để vượt qua, có thể xảy ra hai trường hợp:

2.1) Nếu tất cả các đỉnh của đồ thị được đánh dấu và v là ngẫu nhiên với s , thì sau đó chúng ta đã duyệt qua đồ thị và tìm ra chu trình Hamilton.

2.2) Nếu vẫn còn các đỉnh cần thiết (và vì chúng ta không thể di chuyển đến bất kỳ đỉnh nào trong số chúng), chúng ta tuyên bố giải pháp từng phần hiện tại là vô vọng: nó không thể được mở rộng theo bất kỳ cách nào bằng đỉnh hiện tại. Chúng ta đánh dấu đỉnh cao mà chúng ta đang có và lùi lại một bước để trở lại đỉnh cao mà chúng ta đã đạt được. Chúng ta tiếp tục thử các đỉnh ngẫu nhiên, không được đánh dấu khác, mà chúng ta đã không cố gắng vượt qua cho đến nay.

Nếu trong 2.1) khi tìm một chu trình Hamilton, chúng ta không làm gián đoạn thuật toán, nhưng quay lại, như trong 2.2), thì sự cạn kiệt hoàn toàn sẽ tiếp tục và chúng ta sẽ tìm thấy tất cả các chu trình Hamilton có thể có trong đồ thị.

Phương pháp tìm kiếm quay lui và các tính năng của nó sẽ vẫn là chủ đề chúng ta chú ý trong một vài điểm tiếp theo, nơi chúng ta sẽ xem xét một vài bài toán NP-đầy đủ và toàn diện hơn.

Bài tập

- ▷ 6.7. Vị trí nào trong sơ đồ trên là thích hợp để kiểm tra tính tối ưu (ví dụ, nếu bạn đang tìm một chu trình Hamilton với độ dài tối thiểu)?

6.3.1. Sự thỏa mãn của một hàm Boolean

Bài toán thỏa mãn hàm Boolean, theo một nghĩa nào đó, là bài toán đầy đủ NP "đầu tiên". Nó thường được sử dụng để chứng minh định lý Cook từ 6.2, Bằng cách rút gọn nó để chứng minh tính đầy đủ NP của nhiều bài toán khác. Chúng ta sẽ xem xét bài toán này và giải quyết nó bằng cách tìm kiếm quay lui.

Cho các biến Boolean X_1, X_2, \dots, X_n được cho trước. Bằng $\bar{X}_1, \bar{X}_2, \dots, \bar{X}_n$ chúng ta sẽ biểu thị các phủ định của chúng (nghĩa là X_i là "đúng" khi và chỉ khi \bar{X}_i là "sai"). Chúng ta sẽ sử dụng các phép toán tiêu chuẩn để xây dựng các biểu thức Boolean \wedge và \vee , có nghĩa là logic "và" (kết hợp) và logic "hoặc" (disjunction), cũng như các giá trị 0 và 1 để biểu thị "false" và "true", tương ứng. Chúng ta sẽ cho phép sử dụng tính năng phủ định trên toàn bộ biểu thức (xem định luật De Morgan bên dưới). Đôi khi chúng ta biểu thị sự phủ định bằng dấu \neg đặt trước biểu thức, ví dụ: $\neg(A \vee B)$. Liên kết $A \wedge B$ thường được viết là $A.B$ hoặc thậm chí AB .

Trong Bảng 6.1 kết quả có thể có của các phép toán logic \wedge và \vee trên hai biểu thức Boolean được hiển thị.

x	y	$x \wedge y$	$x \vee y$
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	0

Bảng 6.1. Bảng chân lý

Định nghĩa 6.4. Biểu thức Boolean của các biến X_1, X_2, \dots, X_n được gọi là thỏa mãn nếu có một bộ giá trị "đúng" hoặc "sai" đối với X_1, X_2, \dots, X_n mà giá trị của biểu thức là "true".

Bài toán: Kiểm tra xem một biểu thức có đạt yêu cầu hay không.

Ví dụ, biểu thức $(\bar{X}_2 \vee X_2 \wedge X_1) \wedge \bar{X}_3$ thỏa mãn với $X_1 = 1, X_2 = 0, X_3 = 0$ hoặc $X_1 = 1, X_2 = 1, X_3 = 0$. Kiểm tra xem một biểu thức có thỏa mãn hay không là một bài toán *NP*: nếu một phép gán giá trị cho các biến được đưa ra, chúng ta có thể, như sẽ thấy ngay sau đây, với độ phức tạp đa thức kiểm tra xem nó có phải là một giải pháp hay không, nhưng để tìm ra phép gán này không có thuật toán đa thức nào được biết.

Trước khi chuyển sang lời giải của bài toán, chúng ta sẽ nhắc lại một số yếu tố của logic toán học.

Định nghĩa 6.5. Các hằng số 0 và 1, cũng như các biến và sự phủ định của chúng được gọi là *các chữ*.

Định nghĩa 6.6. Một *dạng biểu thức* hoặc là một biểu thức Boolean trong đó chỉ phép toán "hoặc" có liên quan.

Định nghĩa 6.7. Một biểu thức Boolean trong đó chỉ giải trừ được kết hợp bởi phép toán "và" tham gia được gọi là một biểu thức ở *dạng chuẩn liên hợp*.

Với việc áp dụng danh tính nhiều lần

$$A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$$

bất kỳ biểu thức Boolean nào cũng có thể được biểu diễn ở dạng chuẩn liên hợp.

Các quy tắc hữu ích khác để biến đổi biểu thức được đưa ra bởi các định luật De Morgan:

$$\overline{A \vee B} = \bar{A} \wedge \bar{B}$$

$$\overline{A \wedge B} = \bar{A} \vee \bar{B}$$

Hãy xem một biểu thức ví dụ được viết ở dạng thông thường liên hợp:

$$(X_1 \vee X_4) \wedge (\bar{X}_1 \vee X_2) \wedge (X_1 \vee \bar{X}_3) \wedge (\bar{X}_2 \vee X_3 \vee \bar{X}_4) \wedge (\bar{X}_1 \wedge \bar{X}_2 \wedge \bar{X}_3)$$

Để kiểm tra xem biểu thức có thỏa mãn hay không, chúng ta có thể áp dụng phương pháp tổ hợp sau: chúng ta kiểm tra 2 n số phép gán giá trị có thể có cho các biến và với mỗi phép gán, chúng ta kiểm tra xem biểu thức có thỏa mãn hay không, với độ phức tạp $\Theta(2^n)$. Do đó, thuật toán tìm kiếm quay lui có dạng sau:

Chúng ta khởi tạo $i = 1$.

1) Gán cho biến thứ i giá trị "true" và chuyển sang biến thứ $(i + 1)$ (tiến thêm một bước). Nếu việc gán này không dẫn đến quyết định sau đó, chúng ta quay lại (lùi một bước), gán biến "lie" và thử tiếp tục một lần nữa.

2) Trường hợp giới hạn là khi tất cả các biến được gán một giá trị - sau đó giá trị của toàn bộ biểu thức được tính toán và kiểm tra xem nó có phải là "true" hay không.

Trong quá trình triển khai thuật toán được mô tả, biểu thức Boolean, được rút gọn thành dạng chuẩn liên hợp, sẽ được biểu diễn bằng một mảng hai chiều - một hàng của mảng được so sánh với mỗi hàng không tồn tại. Chúng ta sẽ biểu diễn các biến Boolean của một hàng (không tồn tại) với các chỉ số của chúng - số nguyên từ 1 đến n và để phủ định biến X_i , chúng ta sẽ sử dụng số $-i$. Các giá trị values[] chứa các giá trị (0 hoặc 1) mà chúng ta đã gán cho mỗi biến. Hàm true() tính toán giá trị của biểu thức Boolean cho các phép gán được thực hiện trong các values[]. Việc kiểm tra này rất dễ thực hiện: Để toàn bộ biểu thức có giá trị là "true", thì mỗi phép toán bù quân phải có giá trị là "đúng" (vì phép toán giữa các lùn bù quân là "và"). Để một từ không tồn tại có giá trị "đúng", ít nhất một trong các biến trong đó phải có giá trị "đúng" hoặc phủ định của một biến có giá trị "sai" (vì tất cả các ký tự trong một từ không tồn tại đều là kết hợp với phép toán "hoặc").

Việc gán giá trị của các biến trong thuật toán trên được thực hiện bởi hàm gán (không dấu i), và trong trường hợp biến ($i = n$), hàm true () được gọi để kiểm tra xem biểu thức có thỏa mãn hay không (nghĩa là kiểm tra xem việc chiếm đoạt có phải là một giải pháp cho vấn đề). Nhận thức đầy đủ như sau:

Chương trình 6.1. Biểu thức boolean (601bool.c)

```
#include <stdio.h>
```

```

#define MAXN 100 /* Số lượng tối đa biến boolean*/
#define MAXK 100 /* Số lượng biểu thức hoặc*/
const unsigned n = 4; /* Số lượng các biến boolean*/
const unsigned k = 5; /* Số lượng biểu thức hoặc */
const int expr[][]MAXK] = {
    { 1, 4 },
    { -1, 2 },
    { 1, -3 },
    { -2, 3, -4 },
    { -1, -2, -3 }
};

int values[MAXN];

void printAssignment(void)
{ unsigned i;
printf("Biểu thức thỏa mãn cho: ");
for (i = 0; i < n; i++) printf("%u=%u ", i+1, values[i]);
printf("\n");
}

/* ít nhất một chữ phải có giá trị "đúng" trong mỗi biểu thức hoặc */
int true(void)
{ unsigned i;
for (i = 0; i < k; i++) {
    unsigned j = 0;
    char ok = 0;
    while (expr[i][j] != 0) {
        int p = expr[i][j];
        if ((p > 0) && (1 == values[p-1])) { ok = 1; break; }
        if ((p < 0) && (0 == values[-p-1])) { ok = 1; break; }
        j++;
    }
    if (!ok) return 0;
}
return 1;
}

/* Gán giá trị cho các biến*/

```

```

void assign(unsigned i)
{ if (i == n) {
    if (true()) printAssignment();
    return;
}
values[i] = 1; assign(i + 1);
values[i] = 0; assign(i + 1);
}

int main()
{
    assign(0);
    return 0;
}

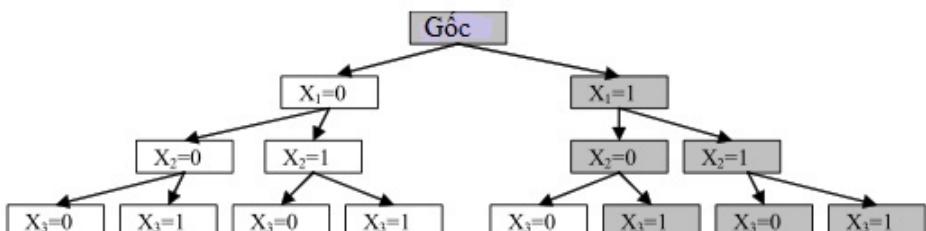
```

Kết quả thực hiện chương trình:

Biểu thức thỏa mãn cho: $X_1 = 1 \ X_2 = 1 \ X_3 = 0 \ X_4 = 0$

Biểu thức thỏa mãn: $X_1 = 0 \ X_2 = 0 \ X_3 = 0 \ X_4 = 1$

Chúng ta sẽ minh họa cách thực hiện phép gán trong một ví dụ với 3 biến Boolean: Hình 6.2 - các đỉnh dẫn đến thỏa mãn biểu thức $X_1 \wedge (X_2 \vee X_3)$ được đánh dấu đậm hơn.



Hình 6.2. Thỏa mãn của biểu thức boolean

Trong các bài toán *NP*-đầy đủ (và đặc biệt trong bài toán thỏa mãn), các ứng cử viên cho giải pháp mặc nhiên tạo thành một cây (hoặc đồ thị) mà thuật toán tìm kiếm quay lui đi qua. Tuy nhiên, thông thường, nghiên cứu về phần lớn loài cây này là không cần thiết. Ví dụ, đối với biểu thức $X_1 \wedge (X_2 \vee X_3)$ khi gán $X_1 = 0$, ngay lập tức nhận thấy rằng chúng ta sẽ không đạt được một nghiệm, bất kể các giá trị sẽ nhận hai biến khác X_2 và X_3 . Khi các biến nhiều hơn

và biểu thức Boolean phức tạp hơn, việc cắt cây được nghiên cứu như vậy có thể làm tăng đáng kể tốc độ của chương trình.

Chúng ta sẽ thực hiện việc cắt với sự trợ giúp của các sửa đổi sau trong chương trình trên:

1) Một đối số t mới được thêm vào hàm true(), nghĩa là chỉ các biến t đầu tiên mới được gán giá trị. Do đó, nếu một từ không tồn tại chỉ bao gồm các biến có số nhỏ hơn hoặc bằng t và không có biến nào trong số chúng có giá trị là "true", thì toàn bộ biểu thức Boolean sẽ là "false" và không có ý nghĩa gì khi gán giá trị Cho những biến nt khác. Nếu có các biến trong liên kết chưa được gán giá trị, thì giá trị của nó vẫn chưa được xác định và quá trình gán vẫn tiếp tục.

2) Trong quá trình tạo, hàm true() được gọi ở đầu mỗi bước. Do đó, việc tạo thêm sẽ bị gián đoạn ngay khi nó trở nên vô nghĩa - giá trị của biểu thức Boolean sẽ là một "lời nói dối", bất kể các phép gán khác sẽ được thực hiện như thế nào.

Chương trình 6.2. Biểu thức boolean (602boolcut.c)

```
#include <stdio.h>
/* Số lượng lớn nhất biến boolean */
#define MAXN 100
/* Số lượng lớn nhất biểu thức hoặc*/
#define MAXK 100
const unsigned n = 4; /* Số lượng các biến boolean*/
const unsigned k = 5; /* Số lượng biểu thức hoặc*/
const int expr[][MAXK] = {
    { 1, 4 },
    { -1, 2 },
    { 1, -3 },
    { -2, 3, -4 },
    { -1, -2, -3 }
};

int values[MAXN];

void printAssign(void)
{ unsigned i;
    printf("Biểu thức thỏa mãn với: ");
    for (i = 0; i < n; i++) printf("X%d=%d ", i + 1, values[i]);
    printf("\n");
}
```

```

}

/* ít nhất ký tự phải có giá trị "đúng" trong mọi biểu thức hoặc */
int true(int t)
{ unsigned i;
  for (i = 0; i < k; i++) {
    unsigned j = 0;
    char ok = 0;
    while (expr[i][j] != 0) {
      int p = expr[i][j];
      if ((p > t) || (-p > t)) { ok = 1; break; }
      if ((p > 0) && (1 == values[p-1])) { ok = 1; break; }
      if ((p < 0) && (0 == values[-p-1])) { ok = 1; break; }
      j++;
    }
    if (!ok) return 0;
  }
  return 1;
}

/* Gán giá trị cho các biến*/
void assign(unsigned i)
{ if (!true(i)) return;
  if (i == n) {
    printAssign();
    return;
  }
  values[i] = 1; assign(i + 1);
  values[i] = 0; assign(i + 1);
}

int main()
{
  assign(0);
  return 0;
}

```

Bài tập

- 6.8. Hãy xuất các tiêu chí khác đối với việc loại cây của người đề cử cho lời giải.

- **6.9.** Biểu thức $(\bar{X}_2 \vee X_3 \wedge X_1) \wedge X_2 \wedge (\bar{X}_1 \vee \bar{X}_3)$ có thỏa mãn không?
- **6.10.** 3. Chứng minh đồng dạng $A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$ bằng bảng chân trị.
- **6.11.** Hãy chứng minh các định luật De Morgan.
- **6.12.** Sử dụng định luật De Morgan để chứng minh đẳng thức $A \vee B = \overline{\overline{A} \wedge \overline{B}}$.
- **6.13.** Phép bình đẳng từ bài toán trước thể hiện sự tách rời thông qua phép liên kết và phép phủ định. Đề xuất và chứng minh một công thức thể hiện sự liên kết thông qua phép loại bỏ và phủ định.
- **6.14.** Sử dụng hai bài toán trước, chứng minh rằng các tập $\{\neg, \wedge\}$ và $\{\neg, \vee\}$ có cùng biểu thức là $\{\neg, \wedge, \vee\}$.
- **6.15.** Có đúng là tập $\{\wedge, \vee\}$ có cùng tính biểu cảm với $\{\neg, \wedge, \vee\}$ không?

6.3.2. Tô màu đồ thị

Trong ?? chúng ta đã xác định vấn đề tô màu tối thiểu của một đồ thị. Nó thuộc về lớp NP - khó và trong đoạn này chúng ta sẽ giải quyết nó bằng cách sử dụng phương pháp tìm kiếm quay lui. Để dàng nhận thấy mối liên hệ giữa bài toán khó NP đối với việc tô màu tối thiểu và bài toán NP -đầy đủ để kiểm tra xem biểu đồ có thể tô màu r hay không:

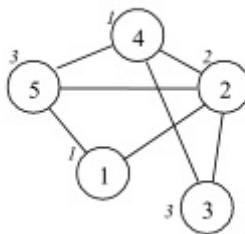
Hãy biểu thị số đỉnh trong đồ thị bằng n . Với mỗi r ($1 \leq r \leq n$) chúng ta sẽ kiểm tra xem đồ thị có thể tô được r màu hay không. Do đó, lời giải của bài toán tô màu tối thiểu là rút gọn đa thức thành tô màu r của một đồ thị.

Hãy xem xét thuật toán sau để kiểm tra khả năng nhuộm r :

- Ta bắt đầu tô màu từ đỉnh đầu tiên của đồ thị: $i = 1$.
- *Bước i:* Tô màu đỉnh thứ i của biểu đồ với màu đầu tiên và chuyển sang tô màu đỉnh $(i + 1)$, sau đó tô màu đỉnh thứ i của biểu đồ bằng màu thứ hai và chuyển sang tô màu $(i + 1)$, v.v.
- Nếu tất cả các đỉnh đều được tô màu (tức là $i = n$) thì ta rơi vào trường hợp biên. Chúng ta kiểm tra xem có các đỉnh lân cận cùng màu hay không và nếu không, chúng ta đã tìm ra giải pháp.

Dễ dàng nhận thấy rằng thuật toán trên cực kỳ kém hiệu quả: độ phức tạp của nó là $\Theta\left(\sum_{r=1}^n r^n\right) = \Theta(n^n)$. Bài toán tương tự như trong bài toán thỏa mãn và như sau: nếu chúng ta tô màu hai đỉnh liền kề có cùng màu ngay từ đầu, chúng ta sẽ tiếp tục tô màu, mặc dù chắc chắn sẽ không dẫn đến giải pháp. Thiếu sót này của thuật toán có thể được khắc phục nếu, khi tô màu cho đỉnh thứ i , chúng ta kiểm tra xem một trong những hàng xóm của nó không còn được tô cùng màu hay không. Nếu trường hợp này xảy ra, thì thay vì đi đến đỉnh $(i + 1)$, chúng ta sẽ cố gắng tô màu thứ i bằng một màu khác. Nếu chúng ta không thể tìm thấy màu cho đỉnh thứ i trong số các màu r , chúng ta lùi lại một bước. Trong kết quả này, nếu $i = 1$ thì việc tô màu của biểu đồ là không thể thực hiện được (chúng ta cần tăng số lượng màu).

Sau đây là mã nguồn của chương trình. Đồ thị được biểu diễn bằng ma trận lân cận $A[][],$ màu của các đỉnh được ghi vào mảng $col[]$ và hàm đệ quy cố gắng tô màu cho đỉnh thứ i của đồ thị là $nextCol(unsigned i)$. Dữ liệu đầu vào mẫu được thể hiện trong Hình 6.3.2.



Hình 6.3. Màu r tối thiểu của đồ thị.

Chương trình 6.3. Tô màu ít nhất của đồ thị (603colorm2.c)

```
#include <stdio.h>
/* Số lượng lớn nhất trong đồ thị */
#define MAXN 200
/* Số đỉnh trong đồ thị */
const unsigned n = 5;
/* Ma trận kề của đồ thị */
const char A[MAXN][MAXN] = {
```

```
{ 0, 1, 0, 0, 1 },
{ 1, 0, 1, 1, 1 },
{ 0, 1, 0, 1, 0 },
{ 0, 1, 1, 0, 1 },
{ 1, 1, 0, 1, 0 });

unsigned col[MAXN], maxCol, count = 0;
char foundSol = 0;

void showSol(void)
{ unsigned i;
count++;
printf("Tô màu ít nhất của đồ thị: \n");
for (i = 0; i < n; i++)
    printf("Đỉnh %u - với màu %u \n", i + 1, col[i]);
}

void nextCol(unsigned i)
{ unsigned k, j, success;
if (i == n) { showSol(); return; }
for (k = 1; k <= maxCol; k++) {
    col[i] = k;
    success = 1;
    for (j = 0; j < n; j++)
        if (1 == A[i][j] && col[j] == col[i]) {
            success = 0;
            break;
        }
    if (success) nextCol(i + 1);
    col[i] = 0;
}
}

int main()
{ unsigned i;
for (maxCol = 1; maxCol <= n; maxCol++) {
    for (i = 0; i < n; i++) col[i] = 0;
    nextCol(0);
    if (count)
        break;
}
```

```

    }
    printf("Tổng số màu được tìm thấy với %u màu: %u \n",maxCol,
           count);
    return 0;
}

```

Kết quả thực hiện chương trình:

Màu tối thiểu của đồ thị:

Đỉnh 1 - với màu 1

Đỉnh 2 - với màu 2

Đỉnh 3 - với màu 3

Đỉnh 4 - với màu 1

Đỉnh 5 - với màu 3

Màu tối thiểu của đồ thị:

Đỉnh 1 - với màu 1

Đỉnh 2 - với màu 3

Đỉnh 3 - với màu 2

Đỉnh 4 - với màu 1

Đỉnh 5 - với màu 2

Màu tối thiểu của đồ thị:

Đỉnh 1 - với màu 2

Đỉnh 2 - với màu 1

Đỉnh 3 - với màu 3

Đỉnh 4 - với màu 2

Đỉnh 5 - với màu 3

Màu tối thiểu của đồ thị:

Đỉnh 1 - với màu 2

Đỉnh 2 - với màu 3

Đỉnh 3 - với màu 1

Đỉnh 4 - với màu 2

Đỉnh 5 - với màu 1

Màu tối thiểu của đồ thị:

Đỉnh 1 - với màu 3

Đỉnh 2 - với màu 1

Đỉnh 3 - với màu 2

Đỉnh 4 - với màu 3

Đỉnh 5 - với màu 2

Màu đồ thị tối thiểu:

Đỉnh 1 - với màu 3

Đỉnh 2 - với màu 2

Đỉnh 3 - với màu 1

Đỉnh 4 - với màu 3

Đỉnh 5 - với màu 1

Tổng số màu được tìm thấy với 3 màu: 6

Bài tập

- ▷ 6.16. Hãy đề xuất một thuật toán để tô màu tối thiểu các đỉnh của một đồ thị, tức là tìm số lượng màu tối thiểu một cách trực tiếp, mà không cần kiểm tra tuần tự từng r ($1 \leq r \leq n$) xem đồ thị có thể tô được r màu hay không.
- ▷ 6.17. Đề xuất thêm tiêu chuẩn loại cây của đăng ký ra quyết định.
- ▷ 6.18. Đề xuất và thực hiện một thuật toán để tô màu tối thiểu các cạnh của đồ thị.

6.3.3. Đường đi đơn dài nhất trong đồ thị chu trình

Cho đến nay, chúng ta đã xem xét một số cách giải thích vẫn đề tìm đường đi đơn giản dài nhất trong đồ thị tuần hoàn có trọng số định hướng (độ dài đường đi được tính bằng tổng các cạnh mà nó chứa). Dưới đây chúng ta sẽ đưa ra một thuật toán của nguyên tắc tìm kiếm quay lui.

Ở mỗi bước, chúng ta cố gắng mở rộng con đường đã xây dựng một phần với một đỉnh khác. Gọi j là đỉnh cuối cùng của con đường được xây dựng cho đến nay. Đối với đường, chúng ta thêm liên tiếp mỗi đỉnh kề với j và không tham gia vào đường. Nếu tại một thời điểm nào đó, chúng ta không thể tìm thấy một đỉnh như vậy, thì điều đó có nghĩa là chúng ta đang ở trong một trường hợp ranh giới. Sau đó, chúng ta tính toán chiều dài của con đường và nếu nó lớn hơn mức tối đa tìm được cho đến nay, chúng ta lưu nó:

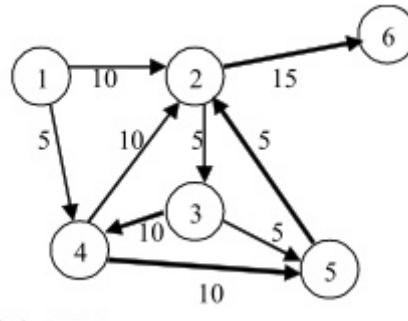
Sơ đồ thêm đỉnh

```
void themmotdinh(i) {
    if (<ta tìm theo đường dài>) {<ta lưu nó và chấm dứt nó>; }
    for (<mọi đỉnh k thừa kế của i>)
        if ((<đường không chứa đỉnh k>) &&
            (k là đỉnh kề của đỉnh tiếp theo trên đường>))
```

```
{  
    <Thêm đỉnh k vào đường>;  
    <Thêm độ dài của cạnh mới vào độ dài của đường>;  
    themmotdinh(i+1);  
    /* Quay lui hồi quy */  
    <Bỏ đánh dấu k là tham gia trong đường>;  
    <Xóa đỉnh k từ đường>;  
    <Trừ độ dài đường với độ dài cạnh mới thêm vào>;  
}  
}
```

Việc thực hiện theo sơ đồ trên. Tuy nhiên, nó không chỉ định đâu là các đỉnh bắt đầu. Rõ ràng là chỉ có đỉnh không có các cạnh đi trước thì không thể chọn làm điểm xuất phát (khi có các cạnh âm thì có thể bỏ sót một giải pháp tối ưu). Hay, việc tìm kiếm phải bắt đầu từ mọi đỉnh cao nhất có thể. Trong phần triển khai bên dưới, chúng ta sẽ gọi `addVertex()` tuân tự cho mỗi đỉnh của đồ thị.

Đồ thị được biểu diễn bằng ma trận lân cận $A[][]$, và các đầu vào mẫu được thể hiện trong Hình 6.4.



Hình 6.4. Đường dẫn đơn dài nhất trong một đồ thị có định hướng.

Chương trình 6.4. Tìm đường dài nhất (604longpath.c)

```
#include <stdio.h>
/* Số lượng lớn nhất của đỉnh */
#define MAXN 200
/* Số lượng đỉnh trong đồ thị*/
const unsigned n = 6;
```

```

/* Ma trận kè của đồ thị */
const char A[MAXN][MAXN] = {
    { 0, 10, 0, 5, 0, 0 },
    { 0, 0, 5, 0, 0, 15 },
    { 0, 0, 0, 10, 5, 0 },
    { 0, 10, 0, 0, 10, 0 },
    { 0, 5, 0, 0, 0, 0 },
    { 0, 0, 0, 0, 0, 0 };
}

unsigned vertex[MAXN], savePath[MAXN];
char used[MAXN];
int maxLen, tempLen, si, ti;

void addVertex(unsigned i)
{ unsigned j, k;
    if (tempLen > maxLen) { /*Đa tìm đường đường dài nhất và lưu lại */
        maxLen = tempLen;
        for (j = 0; j <= ti; j++) savePath[j] = vertex[j];
        si = ti;
    }
    for (k = 0; k < n; k++) {
        if (!used[k]) { /*Nếu đỉnh k không tham gia vào đường đến hiệ
n tại*/
            /* Nếu đỉnh như vậy thì thêm vào và là bên cạnh của đường
cuối */
            if (A[i][k] > 0) {
                tempLen += A[i][k];
                used[k] = 1; /* đánh dấu k đã tham gia vào đường */
                vertex[ti++] = k; /* thêm đỉnh k vào đường*/
                addVertex(k);
                used[k] = 0; /* Quay lui hồi quy */
                tempLen -= A[i][k]; ti--;
            }
        }
    }
}

int main() {
    unsigned i;

```

```

maxLen = 0; tempLen = 0; si = 0; ti = 1;
for (i = 0; i < n; i++) used[i] = 0;
for (i = 0; i < n; i++) {
    used[i] = 1; vertex[0] = i;
    addVertex(i);
    used[i] = 0;
}
printf("Đường dài nhất là: \n");
for (i = 0; i < si; i++) printf("%u ", savePath[i] + 1);
printf("\nvới độ dài chung %d\n", maxLen);
return 0;
}

```

Kết quả thực hiện chương trình:

Cách dài nhất là:

3 4 5 2 6

với tổng chiều dài là **40**.

Bài tập

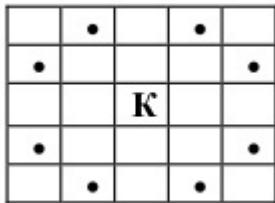
▷ **6.19.** Bài toán tìm đường đi đơn dài nhất trong đồ thị tuần hoàn có trọng số có định hướng có thay đổi về nguyên tắc không nếu chiều dài đường đi được tính bằng tích các trọng lượng của các sườn mà nó chứa. Và nếu chiều dài được xác định bởi trọng lượng của xương sườn tối thiểu (tối đa) liên quan đến đường?

▷ **6.20.** Bài toán tìm đường đi nguyên tố dài nhất trong đồ thị tuần hoàn có định hướng có thay đổi về nguyên tắc không, nếu độ dài đường đi được tính bằng tổng trọng số của các đỉnh mà nó chứa (chúng ta giả sử rằng đối với mỗi đỉnh thì một trọng số - a số tự nhiên).

▷ **6.21.** Sửa đổi chương trình trên để in tất cả các đường có độ dài tối đa.

6.3.4. Đường đi quan ngựa

Một tác vụ phổ biến minh họa tốt cho phương pháp tìm kiếm trả về được gọi là "đi bộ ngựa":



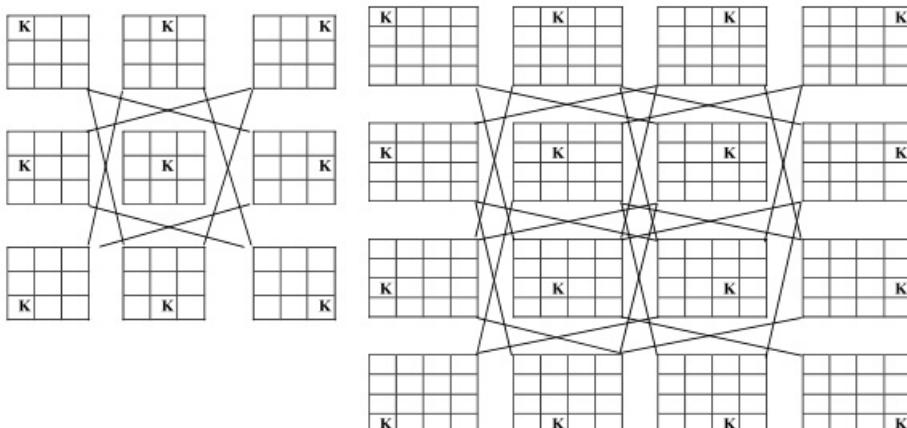
Hình 6.5. Vị trí quân ngựa
được di chuyển.

8	22	7	44	39	24	9	28	63
7	43	40	23	8	45	62	25	10
6	6	21	42	59	38	27	64	29
5	41	58	37	46	61	54	11	26
4	20	5	60	53	36	47	30	51
3	57	2	35	48	55	52	15	12
2	4	19	56	33	14	17	50	31
1	1	34	3	18	49	32	13	16
	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>	<i>h</i>

Hình 6.6. Lời giải bài toán với $n = 8$.

Bài toán: Cho một số tự nhiên n ($n > 4$) và một bàn cờ tổng quát có kích thước $n \times n$ ô. Tìm một con bò trên bảng với đường đi của con ngựa. Mỗi ô trên bảng phải được truy cập chính xác một lần, với "chuyển tham quan" bắt đầu từ ô ở góc dưới bên trái. Nước đi cho phép của ngựa là nước cờ tiêu chuẩn, như trong Hình 6.5.

Lời giải của bài toán cho $n = 8$ được thể hiện trong Hình 6.6. Tại mỗi vị trí của con ngựa trên bàn cờ (tổng cộng là n^2) một đỉnh của đồ thị vô hướng có thể được so sánh. Một đường sườn trong đồ thị này sẽ nối hai đỉnh, nếu có thể chuyển đổi giữa các vị trí tương ứng bằng hành trình của con ngựa. Đây là những đồ thị tương tự cho $n = 3$ và $n = 4$ trông như thế nào (Hình 6.7):



Hình 6.7. Đồ thị bài toán với $n = 3$ và $n = 4$

Để tìm đường đi ngang cần thiết của bàn cờ, chúng ta phải tìm một đường đi trong đồ thị mà mỗi đỉnh đi đúng một lần, tức là đường đi của Hamilton. Vì vậy, chúng ta nhận được điều đó

"Bài toán nước đi quân ngựa" ("Đường Hamilton"

Hơn nữa, bài toán ngựa đi bộ có thể được rút gọn thành bài toán tìm đường đi Hamilton trong đồ thị trong đó hoành độ của mỗi đỉnh di bị giới hạn, trong trường hợp này là $d_i \leq 8$ (xem các bài toán 5, 7 và 8 ở cuối phần này).

Tất nhiên, chúng ta sẽ không giải quyết vấn đề với một đồ thị, thứ nhất, vì có một thuật toán rõ ràng hơn nhiều, và thứ hai, bởi vì một đồ thị với n^2 đỉnh như vậy sẽ yêu cầu khoảng $8.n^2$ bộ nhớ (để biểu diễn thông qua một danh sách các . thậm chí sẽ cần nhiều bộ nhớ hơn - xem Bảng ??). Một lần nữa, chúng ta sẽ sử dụng tìm kiếm trả về, được mô tả bằng sơ đồ bởi hàm đệ quy sau:

Bước đi của quân ngựa

```
/* board[][] là bàn cờ cỡ n x n */
void BuocTiepTheo(x, y, i)
{ if (i == n*n) { /*in ra nghiệm bài toán* */
    board[x][y] = i;
    for (< mỗi bước có thể đi (u,v) của quân ngựa trên ô (x,y))
        if (0 == board[u][v]) /* Nếu ô đặt là trống */
            BuocTiepTheo(u, v, i+1);
    board[x][y] = 0; /*quay lui */
}
```

Trong quá trình thực hiện sau đó, một bản tóm tắt của bài toán được xem xét. Đầu tiên, chuyến tham quan có thể bắt đầu từ bất kỳ ô nào (đặt trước với tọa độ `startX` và `startY`). Ngoài ra, các nét cho phép đổi với con ngựa có thể được xác định lại, tức là. chúng có thể không phải là những cái tiêu chuẩn được thể hiện trong Hình 6.5, mà là những cái khác ngẫu nhiên.

Các bước di chuyển hợp lệ cho ngựa được đặt ở đầu chương trình trong các mảng không đổi `diffX[]` và `diffY[]` - các giá trị trong đó có nghĩa là ngựa có thể di chuyển từ ô (x, y) sang bất kỳ ô nào khác ($x + \text{diffX}[i]$, $y + \text{diffY}[i]$), cho $i = 1, 2, \dots, maxDiff$:

Chương trình 6.5. Bài toán các bước đi quan ngựa (605knight.c)

```

#include <stdio.h>
#include <stdlib.h>
/* Cỡ lớn nhất của bàn cờ */
#define MAXN 10
/* Số lớn nhất quy tắc di chuyển của quân ngựa*/
#define MAXD 10
/* Cỡ của bàn cờ thật */
const unsigned n = 6;
/* Vị trí bắt đầu */
const unsigned startX = 1;
const unsigned startY = 1;
/* Quy tắc di chuyển của quân ngựa*/
const unsigned maxDiff = 8;
const int diffX[MAXD] = { 1, 1, -1, -1, 2, -2, 2, -2 };
const int diffY[MAXD] = { 2, -2, 2, -2, 1, 1, -1, -1 };
unsigned board[MAXN][MAXN];
unsigned newX, newY;

void printBoard(void)
{ unsigned i, j;
  for (i = n; i > 0; i--) {
    for (j = 0; j < n; j++) printf("%3u", board[i-1][j]);
    printf("\n");
  }
  exit(0); /* thoát khỏi chương trình */
}

void nextMove(unsigned X, unsigned Y, unsigned i)
{ unsigned k;
  board[X][Y] = i;
  if (i == n * n)
    { printBoard(); return; }
  for (k = 0; k < maxDiff; k++) {
    newX = X + diffX[k]; newY = Y + diffY[k];
    if ((newX >= 0 && newX < n && newY >= 0 && newY < n) &&
        (0 == board[newX][newY]))
      nextMove(newX, newY, i + 1);
  }
  board[X][Y] = 0;
}

```

```

    }

int main() {
    unsigned i, j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            board[i][j] = 0;
    nextMove(startX-1, startY-1, 1);
    printf("Bài toán không có nghiệm. \n");
    return 0;
}

```

Kết quả thực hiện chương trình:

```

10 27  6 19 16 25
 7 20  9 26  5 18
28 11  4 17 24 15
21  8 23 32  3 34
12 29  2 35 14 31
 1 22 13 30 33 36

```

Lưu ý: Bài toán đi ngang bàn cờ với một cú đánh của con ngựa thường được coi là một ví dụ kinh điển về việc áp dụng phương pháp tìm kiếm trả về, mặc dù nó không phải là *NP*-đầy đủ. Vì nó có một nghiệm đa thức (theo sau đó là một nghiệm đa thức để tìm đường đi Hamilton trong dạng đặc biệt của đồ thị trong Hình 6.7). Thuật toán này (sẽ được thảo luận chi tiết trong Chương 9 - Thuật toán tham lam) thường dựa trên sơ đồ sau: đối với mỗi nước đi liên tiếp của ngựa, ô này được chọn, từ đó sẽ có ít cơ hội nhất cho nước đi tiếp theo.

Tuy nhiên, với những sửa đổi nhỏ cho bài toán, chúng ta có thể dễ dàng thấy mình rơi vào tình huống mà chúng ta không thể chắc chắn rằng bài toán có phải là đa thức hay không. Ví dụ như vậy là các bài toán sau:

Bài tập

- ▷ 6.22. Hãy sửa đổi chương trình trên để nó tìm thấy tất cả các lời giải.
- ▷ 6.23. Hãy sửa đổi chương trình trên để nó tìm thấy tất cả các giải

pháp bắt đối xứng khác nhau. Chúng ta sẽ xem xét các nghiệm đối xứng có thể nhận được từ nhau bằng cách quay hoặc bằng phép đổi xứng về các đường ngang, đường thẳng đứng hoặc hai đường chéo chính của bàn cờ.

▷ **6.24.** Tìm số đường đi khác nhau của bàn cờ:

- a) tất cả các đường duyệt qua
- (b) tất cả các đường dfuyệt không đối xứng khác nhau

▷ **6.25.** Cho một số tự nhiên $n(n > 4)$ và một bàn cờ tổng quát có kích thước $n \times n$ ô. Tìm một con đường trên bảng với đường đi của con ngựa. Mỗi ô phải được truy cập chính xác một lần, với "chuyển tham quan" bắt đầu từ bất kỳ ô nào được đặt trước trên bàn cờ.

▷ **6.26.** Cho một số tự nhiên $n(n > 4)$ và một bàn cờ tổng quát có kích thước $n \times n$ ô được đưa ra, một số bị cấm. Tìm một con đường trên bảng với bước đi của con ngựa. Mỗi ô không bị cấm trên bàn cờ phải được truy cập chính xác một lần, với "chuyển tham quan" bắt đầu từ ô ở góc dưới bên trái.

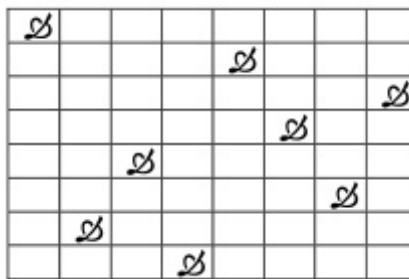
▷ **6.27.** Một số tự nhiên $n(n > 4)$ và một bàn cờ tổng quát có kích thước $n \times n$ ô được đưa ra. Tìm một con đường đi trên bảng với bước đi của quân ngựa với giao điểm tối thiểu của quỹ đạo quân ngựa.

▷ **6.28.** Xác định hạng của bài toán tìm đường đi Hamilton trong đồ thị có tung độ đỉnh giới hạn (tức là tung độ của mỗi đỉnh nhỏ hơn một số tự nhiên k cố định).

▷ **6.29.** Tính toán đa thức bài toán **6.28** thành bài toán **6.26** có được không?

6.3.5. Bài toán tám quân Hậu

Bài toán đặt quân hậu trên bàn cờ là một bài toán kinh điển. Nó một lần nữa cho thấy tầm quan trọng của việc chặt cây đê quy đúng cách đối với việc áp dụng hiệu quả phương pháp triệt tiêu hoàn toàn. Gauss đã xem xét bài toán này ngay từ năm 1850 và đã thành công chính xác do thực tế là với nghiên cứu phân tích, các ứng cử



Hình 6.8. Lời giải bài toán quân Hậu với $n = 8$

viên cho một giải pháp có thể giảm đáng kể. Do đó, mục tiêu của chúng ta sẽ là thu hẹp phạm vi nghiên cứu càng nhiều càng tốt và chỉ sau đó mới thực hiện vét cạn hoàn toàn.

Bài toán: Đặt n quân Hậu trên một bàn cờ tổng quát có kích thước $n \times n$ ($n \geq 2$) sao cho không có hai quân nào bị tác động ăn quân (không nằm trong cùng một hàng ngang, dọc hoặc chéo).

Một lời giải khả thi của bài toán cho $n = 8$ được thể hiện trong Hình 6.8. Chúng ta sẽ giải quyết vấn đề bằng cách tìm kiếm với quay lui theo sơ đồ sau:

Chúng ta khởi tạo $i = 1$.

- Nếu chúng ta đã triển khai thành công các quân Hậu cho đến nay, chúng ta đang cố gắng tìm một vị trí "phù hợp" của quân Hậu ($i + 1$) (bước tiến).
- Nếu chúng ta đã đặt n quân Hậu (tức là $i = n$) và chúng đáp ứng điều kiện của bài toán, thì sau đó chúng ta đã tìm ra lời giải, in ra, và cuối cùng dừng phép tính.
- Nếu không có vị trí trống cho quân hậu thứ i , chúng ta lùi lại một bước và tìm kiếm vị trí khác cho quân hậu (thứ $i - 1$).

Chúng ta sẽ làm rõ ý nghĩa của vị trí "thích hợp" ở trên. Một giải pháp không hiệu quả cho bài toán là thử tất cả các cách kết hợp có thể có $C_{n^2}^n$ để đặt các quân hậu trên bàn cờ và loại bỏ những kết hợp không phải là giải pháp cho bài toán. Vì chỉ có thể có nhiều nhất một quân Hậu trong mỗi hàng, nên chỉ có thể thử n^n vị trí quân hậu khác nhau, một trong mỗi hàng và có thể tìm ra giải pháp cho chúng. Nếu chúng ta tính đến việc có thể có nhiều nhất một

quân Hậu trong một cột, thì số khả năng sinh ra giảm xuống còn $n!$. Chúng ta sẽ giới thiệu một mảng n phần tử chứa số cột - ở vị trí thứ i được viết số cột mà ở đó vị trí quân Hậu của hàng thứ i . Mỗi hoán vị như vậy sẽ xác định vị trí rõ ràng của các quân Hậu trên bàn cờ. Cấu hình xác minh có thể được giảm hơn nữa - chỉ các vị trí an toàn sẽ được chọn khi đặt từng quân Hậu kế tiếp, tức là những vị trí không ở cùng vị trí ngang, dọc hoặc chéo với quân Hậu đã được đặt. Đây là thuật toán được mô tả bằng sơ đồ trông như thế nào:

Đặt Hậu trên hàng

```
void DatHauHang(i) { /* Đặt Hậu hàng thứ i */
    if (i > n) { <In ra Lời giải>; }
    for (<mọi cột k quân Hậu đã chiếm>
        if (<không có hậu trên đường chéo qua (i,k)>) {
            /* vị trí (i,k) không bị đe dọa */
            <Đặt Hậu i ở vị trí (i,k)>;
            DatHauHang(i+1);
            <loại bỏ hậu thứ i từ vị trí (i,k)>;
        }
    }
}
```

Trong quá trình triển khai để đánh dấu và kiểm tra xem một vị trí có an toàn hay không, chúng ta sẽ sử dụng bốn mảng một chiều:

1) `queens[N]`: `queens[i]` chứa số lượng của trụ cột mà quân hậu của hàng thứ i được đặt.

2) `col[N]`: `col[i]` có giá trị là 1 nếu không có quân hậu nào được đặt trong cột i và 0 - ngược lại.

3) `RD[2*N-1]` và `LD[2*N-1]` cho biết liệu có nữ hoàng trên đường chéo chính thứ k và đường chéo phụ tương ứng ($k = 1, 2, \dots, 2n - 1$). Trong đó:

- `RD[i + k]` cho biết có quân hậu trong đường chéo chính đi qua ô (i, k) hay không.
- `LD[N + i-k]` cho biết có quân hậu trong đường chéo phụ đi qua ô (i, k) hay không.

Chương trình dưới đây kết thúc với một giải pháp. Chúng ta cho phép người đọc cố gắng sửa đổi nó để nó tìm và suy ra tất cả các giải pháp có thể cho một n nhất định.

Chương trình 6.6. Bài toán tám quân Hậu (606queens.c)

```

#include <stdio.h>
#include <stdlib.h>
/* Kích thước lớn nhất bàn cờ */
#define MAXN 100
/* Kích thước bàn cờ */
const unsigned n = 13;

unsigned col[MAXN], RD[2*MAXN - 1],
LD[2*MAXN], queens[MAXN];

/* In ra vị trí các quân Hậu */
void printBoard()
{
    unsigned i, j ;
    for (i = 0; i < n; i++) {
        printf("\n");
        for (j = 0; j < n; j++)
            if (queens[i] == j)
                printf("x ");
            else
                printf("o ");
    }
    printf("\n");
    exit(0);
}

/* Tìm vị trí ti ép theo đđ để đặt Hậu */
void generate(unsigned i)
{
    unsigned j;
    if (i == n) printBoard();
    for (j = 0; j < n; j++)
        if (col[j] && RD[i + j] && LD[n + i - j]) {
            col[j] = 0; RD[i + j] = 0; LD[n + i - j] = 0; queens[i] = j;
            generate(i + 1);
            col[j] = 1; RD[i + j] = 1; LD[n + i - j] = 1;
        }
}

int main()
{
    unsigned i;

```

```

for (i = 0; i < n; i++) col[i] = 1;
for (i = 0; i < (2*n - 1); i++) RD[i] = 1;
for (i = 0; i < 2*n; i++) LD[i] = 1;
generate(0);
printf("Bài toán không có nghiệm! \n");
return 0;
}

```

Kết quả thực hiện chương trình:

```

x 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 x 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 x 0 0 0 0 0 0 0 0 0 0
0 x 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 x 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 x 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 x 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 x
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 x 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 x 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 x 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 x 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 x 0 0

```

Các tối ưu hóa bổ sung của thuật toán cũng có thể được thực hiện, tuy nhiên, điều này sẽ không làm giảm độ phức tạp $\Theta(n!)$. Của nó [Nakov-1998] [Reingold, Nivergelt, Deo-1980].

Bài tập

- ▷ 6.30. Đề xuất và thực hiện một thuật toán để giải một dạng bài toán sau: In tất cả các vị trí đối xứng có thể có của các quân Hậu trên bảng. Hai giải pháp được coi là đối xứng nếu có thể nhận được một giải pháp từ phương pháp kia bằng cách xoay bàn cờ hoặc đổi xứng về các hàng, cột, đường chéo chính hoặc phụ.

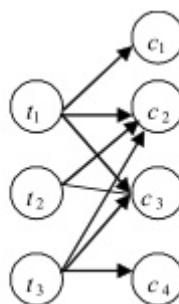
- ▷ 6.31. Bài toán đối với trường hợp tổng quát của một bảng hình chữ nhật có hợp lý không (và nếu có, với những giải thích bổ sung nào)?

6.3.6. Thời khóa biểu của trường học

Lập lịch là một ví dụ khác về một bài toán được giải quyết bằng phương pháp tìm kiếm quay lui và vét cạn hoàn toàn. Nếu không xem xét chi tiết câu hỏi rộng này (người đọc có thể tìm thêm thông tin trong [Christofides-1975]), chúng ta sẽ cố gắng làm rõ nó nói chung bằng cách xem xét một phiên bản đơn giản của bài toán biên soạn chương trình giảng dạy ở trường học:

Bài toán: Lớp C và t giáo viên được đưa ra (c, t - số tự nhiên). Số giờ $c1[i][j]$ (số tự nhiên, $i = 1, 2, \dots, c, j = 1, 2, \dots, t$) mà giáo viên j phải thuyết trình cho lớp i được biết. Để tìm một lịch trình cho chương trình học, trong đó các điều kiện sẽ được đáp ứng:

- 1) Trong mỗi bài học, một giáo viên có thể dạy nhiều nhất một lớp.
- 2) Tổng thời lượng của chương trình phải ở mức tối thiểu.



Hình 6.9. Lịch dạy học

Xét ví dụ sau: Có 4 lớp c_1, c_2, c_3 và c_4 và ba giáo viên t_1, t_2 và t_3 . Các giờ sẽ được tổ chức là:

Giáo viên dạy t_1 mỗi lớp c_1 -5 giờ, t_1 mỗi c_2 - 5 giờ, t_1 mỗi c_3 - 5 giờ,

Giáo viên dạy t_2 trong lớp c_2 - 5 giờ, t_2 trong c_3 - 5 giờ,

Giáo viên dạy t_3 mỗi lớp c_2 - 5 giờ, t_3 mỗi c_3 - 5 giờ, t_3 mỗi c_4 - 5 giờ.

Chúng ta đã trình bày cấu hình này theo sơ đồ với một đồ thị hai phần có định hướng trong Hình 6.9: Chúng ta chưa xác định trọng số của các sườn: đối với ví dụ đã chọn, tất cả các trọng số sẽ bằng 5. Với $t_i \rightarrow c_j(x)$, chúng ta nghĩa là giáo viên dạy bạn x giờ của lớp c_j . Do đó, hai lịch trình sau đây có thể thực hiện được cho ví dụ đã

chọn:

Lựa chọn 1.

- 1) $t_1 \rightarrow c_2 (5), t_3 \rightarrow c_3 (5), t_2$ nghỉ vì cả hai lớp c_2 và c_3 đều bị chiếm.
- 2) $t_1 \rightarrow c_3 (5), t_3 \rightarrow c_2 (5), t_2$ nghỉ.
- 3) $t_2 \rightarrow c_2 (5), t_1 \rightarrow c_1 (5), t_3 \rightarrow c_4 (5)$
- 4) $t_2 \rightarrow c_3 (5)$

Chương trình đã đầy đủ và tổng thời lượng của nó là 20 giờ.

Lựa chọn 2.

- 1) $t_1 \rightarrow c_1 (5), t_2 \rightarrow c_2 (5), t_3 \rightarrow c_3 (5)$
- 2) $t_1 \rightarrow c_2 (5), t_2 \rightarrow c_3 (5), t_3 \rightarrow c_4 (5)$
- 3) $t_1 \rightarrow c_3 (5), t_3 \rightarrow c_2 (5), t_2$ nghỉ.

Đến đây chương trình hoàn thành trong 15 giờ. Có thể thấy rằng tổng thời lượng phụ thuộc vào trình tự dạy học.

Cả hai lựa chọn được thể hiện trong Bảng ??.

Phương án 1.

	1-5	6-10	11-15	16-20
t_1	c_2	c_3	c_1	\times
t_2	\times	\times	c_2	c_1
t_3	c_3	c_2	c_4	\times

Phương án 2.

	1-5	6-10	11-15
t_1	c_2	c_3	c_1
t_2	c_2	c_3	\times
t_3	c_3	c_4	c_2

Bảng 6.2. Các phương án lịch

Như một điểm khởi đầu để giải quyết vấn đề, chúng ta sẽ xem xét thuật toán tổ hợp sau: Để tìm một chương trình có thời lượng tối thiểu, chúng ta sẽ xây dựng và đánh giá tất cả các lịch trình có thể có. Hãy xem bảng sau (Bảng 6.3):

Bảng cho biết việc làm của mỗi giáo viên trong mỗi giờ, tức là trong ô của hàng thứ i, cột thứ j sẽ ghi lớp học mà giáo viên thứ i sẽ dạy trong giờ thứ j. Chúng ta sẽ điền vào các cột của bảng một cách

	1	2	3	4	5	6	7	8	...
t_1									
t_2									
...									

Bảng 6.3. Bảng phân công

nhất quán với tất cả các tổ hợp lớp có thể có để đáp ứng các điều kiện của bài toán:

- Tại một thời điểm, mỗi giáo viên có thể dạy nhiều nhất một lớp (nghĩa là không thể có sự lặp lại các lớp trong một cột).
- Mỗi giáo viên nên dành chính xác bao nhiêu giờ cho lớp học tương ứng theo điều kiện quy định.

Ví dụ, đối với *Lựa chọn 2* ở trên, bảng sẽ được điền như trong Bảng 6.4.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
t_1	c_1	c_1	c_1	c_1	c_1	c_2	c_2	c_2	c_2	c_2	c_3	c_3	c_3	c_3	c_3
t_2	c_2	c_2	c_2	c_2	c_2	c_3	c_3	c_3	c_3	c_3	\times	\times	\times	\times	\times
t_3	c_3	c_3	c_3	c_3	c_3	c_4	c_4	c_4	c_4	c_4	c_2	c_2	c_2	c_2	c_2

Bảng 6.4. Bảng phân công

Quyết định điền vào một bảng như vậy sẽ không hiệu quả, vì chúng ta tạo ra một lịch trình, xem xét từng giờ riêng biệt. Trong ví dụ này, tất cả giáo viên dành 5 giờ - vì vậy chúng ta có thể tạo lịch trình cho mỗi 5 giờ liên tục (Bảng ??):

	5	10	15	20	...
t_1					
t_2					
...					

Bảng 6.5. Bảng phân công

Trong trường hợp chung, vì giờ của mỗi giáo viên là số tự nhiên ngẫu nhiên nên chúng ta sẽ thực hiện như sau ở mỗi bước:

- Ta tính $mX = \min(cl[i][j])$, với mỗi $i = 1, 2, \dots, c, j = 1, 2, \dots, t, cl[i][j] \neq 0$.
- Chúng ta tạo ra tất cả các kết hợp có thể có của các bài tập trên lớp cho mỗi giáo viên trong khoảng thời gian mX giờ: mỗi giáo viên $t_i \rightarrow$ lớp c_j , với mỗi $i = 1, 2, \dots, t$, và đối với mỗi bài tập, chúng ta sẽ giảm đi mX giờ $cl[i][j]$.

Quá trình tạo tiếp tục cho đến khi tất cả các giá trị của $cl[i][j]$ bằng 0 - khi đó sẽ có một trường hợp giới hạn trong đó nó được tính toán thời lượng chương trình kéo dài và nếu số kết quả nhỏ hơn thời lượng nhỏ nhất được tìm thấy cho đến nay, lịch trình được duy trì.

Chúng ta sẽ triển khai thuật toán dưới dạng một hàm `generate(teacher, level, mX, totalHours)` với 4 tham số: biến `teacher` chứa số giáo viên mà một lớp sẽ được chỉ định, `mX` là số giờ và `totalHours` - tổng thời lượng của chương trình cho đến nay. Đối với mỗi mức (đòng điện được giữ ở `level`), đầu tiên `mX` được tính theo thuật toán mô tả ở trên, sau đó `generate(0, level+1, mX, totalHours+mX)`. Điều này có nghĩa là trong tương lai, một lớp học phải được chỉ định cho mỗi giáo viên để dạy số giờ `mX`. Việc giao lớp `k` cho giáo viên được thực hiện theo thuật toán cho các tổ hợp không lặp lại (xem ??). Sau khi phân công, số giờ dạy `cl[k][teacher]` giảm đi `mX`, và khi một lớp được chỉ định cho mỗi giáo viên (`teacher == N`), lớp đó sẽ được chuyển sang cấp độ tiếp theo và toàn bộ quá trình được lặp lại từ đầu - `mX` được tính toán lại, v.v. n. Phần cuối của đệ quy đạt được khi không còn giờ dạy - sau đó nó được kiểm tra xem `totalHours < minimal` (thời lượng của chương trình mới được tạo có nhỏ hơn thời lượng tối thiểu cho đến thời điểm này hay không) và nếu hoàn thành, lịch trình sẽ được lưu.. . Đệ quy cũng bị cắt:

```
if (totalHours >= Minimum) return;
```

nghĩa là, nếu thời lượng của chương trình được tạo một phần trở nên dài hơn thời lượng tối thiểu được tìm thấy cho đến nay, thì việc tiếp tục tạo sẽ không có ý nghĩa gì.

Chương trình 6.7. Lập thời khóa biểu (607program.c)

```

#include <stdio.h>
/* Số giáo viên lớn nhất */
#define MAXT 100
/* Số lời lớn nhất */
#define MAXC 100
/* Số giáo viên */
const unsigned t = 3;

/* Số lớp */
const unsigned c = 4;
unsigned cl[MAXC][MAXT] = {
    { 5, 5, 5 }, /* thông tin lớp 1 */
    { 5, 5, 5 },
    { 5, 0, 0 },
    { 0, 0, 5 } /* thông tin lớp C */
};

const unsigned MAX_VALUE = 20000;

char usedC[100][MAXC];
unsigned teach[MAXT], minimal;

void generate(unsigned teacher, unsigned level,
              unsigned mX, unsigned totalHours)
{
    unsigned i, j;
    if (totalHours >= minimal) return;
    if (teacher == t) {
        unsigned min = MAX_VALUE;
        for (i = 0; i < c; i++)
            for (j = 0; j < t; j++)
                if (cl[i][j] < min && 0 != cl[i][j]) min = cl[i][j];
        if (min == MAX_VALUE) {
            if (totalHours < minimal) minimal = totalHours;
        }
        else {
            generate(0, level + 1, min, totalHours + min);
        }
    }
    return;
}

```

```

/* Xác định lớp cho giáo viên dạy, mà sử dụng giờ nhỏ nhất*/
for (i = 0; i < c; i++) {
    if (cl[i][teacher] > 0 && !usedC[level][i]) {
        cl[i][teacher] -= mX;
        usedC[level][i] = 1;
        generate(teacher + 1, level, mX, totalHours);
        usedC[level][i] = 0; /* quay lui */
        cl[i][teacher] += mX;
    }
}

int main() {
    unsigned i, j;
    for (i = 0; i < 100; i++)
        for (j = 0; j < c; j++) usedC[i][j] = 0;
    minimal = MAX_VALUE;
    generate(t, 0, 0, 0);
    printf("Lịch trình tối thiểu là %u giờ.\n", minimal);
    return 0;
}

```

Kết quả thực hiện chương trình:

Lịch trình tối thiểu là 15 giờ.

Bài tập

- ▷ 6.32. Sửa đổi chương trình trên sao cho in bản phân phối để dạy học cùng với thời lượng tối thiểu của chương trình.
- ▷ 6.33. Độ phức tạp của thuật toán được đề xuất trên là gì?

6.3.7. Dịch mật mã

Mã Morse thường được sử dụng, mặc dù cũ hơn một chút, như một phương tiện liên lạc - trong điện báo, tàu thủy, để liên lạc giữa các địa điểm quân sự và những nơi khác. Trong đó, mỗi ký hiệu (chữ cái, số, v.v.) được biểu diễn dưới dạng một chuỗi các dấu chấm và dấu gạch ngang. Do đó, biểu tượng có thể được truyền bằng tín hiệu radio (hoặc ánh sáng) ngắn hơn và dài hơn tương ứng với dấu chấm và dấu gạch ngang. Điều quan trọng là quá trình truyền giữa

các ký tự phải được tạm dừng, nếu không, bản dịch ngược lại rất mơ hồ, ví dụ:

Chữ "A" tương ứng với "01" (0 sẽ chỉ dấu chấm, và 1 - dấu gạch ngang), chữ "M" tương ứng với "11" và chữ "J" - "0111". Do đó, nếu nhận được dãy mã "0111", chúng ta sẽ không biết làm thế nào để giải mã nó - là "J" hoặc là "AM".

Bài toán chúng ta sẽ xem xét như sau: Hai bảng chữ cái được đưa ra và đối với mỗi chữ cái của một bảng chữ cái, chúng ta biết chữ cái nào của bảng kia tương ứng với nó ("các chữ cái" của cả hai bảng chữ cái có thể đại diện cho nhiều hơn một ký hiệu). Ví dụ: nếu các ký hiệu tạo thành các chữ cái của bảng chữ cái đầu tiên là 0 và 1, và của thứ hai - a, b, c, d , thì chúng ta có thể xem xét hai bảng chữ cái sau:

- 1) "0", "1", "01", "10"
- 2) " a ", " b ", " c ", " d "

Tất cả các chuỗi này được coi là các chữ cái và mỗi chữ cái của một bảng chữ cái được so sánh duy nhất với một chữ cái khác, ví dụ:

$$\begin{aligned} "0" &\Leftrightarrow "a" \\ "1" &\Leftrightarrow "b" \\ "01" &\Leftrightarrow "c" \\ "10" &\Leftrightarrow "d" \end{aligned}$$

Theo một từ (chuỗi các chữ cái) từ một bảng chữ cái, chúng ta đang tìm kiếm bản dịch sang ngôn ngữ của bảng chữ cái kia. bài toán phức tạp là chúng ta không có dấu phân cách giữa các chữ cái riêng lẻ. Do đó, chuỗi "0110" có thể được dịch theo một số cách: là " $abba$ ", " cd " và các cách khác. Tất cả các bản dịch có thể có của một từ đều được tìm kiếm.

Bài toán được đặt ra đòi hỏi vét canh hoàn toàn các khả năng, vì vậy chúng ta sẽ áp dụng tìm kiếm quay lại. Hãy để từ mà chúng ta đang dịch được viết trong chuỗi str1. Chúng ta sẽ sử dụng tham số count, cho biết ký hiệu mà chúng ta đã đạt được cho đến nay (số đếm ban đầu có giá trị là 0). Chúng ta tìm thấy một chuỗi con bắt đầu bằng số đếm, phải là một chữ cái của bảng chữ cái đầu tiên.

Có thể tìm thấy một số chữ cái như vậy (ví dụ, ở trên trong chuỗi "0110", chúng ta sẽ tìm thấy các chữ cái "0" và "01"). Đối với mỗi ký tự như vậy, chúng ta kiểm tra đệ quy (dịch) phần còn lại của chuỗi sau khi loại trừ nó. Trên thực tế, chúng ta sẽ không sửa đổi chuỗi trong thực tế, nhưng sẽ tăng số lượng theo độ dài của ký tự tìm được. Nếu tại một thời điểm nào đó giá trị của số đếm trở nên bằng độ dài của chuỗi, thì có một trường hợp giới hạn - chúng ta đã nhận được một bản dịch khả dĩ.

Trong quá trình thực hiện, dữ liệu đầu vào được khởi tạo trong hàm initLanguage(). Số chữ cái là N , trường của mảng $\text{transf}[i].\text{st1}$ chứa chữ cái thứ i ($1 \leq i \leq n$) của bảng chữ cái đầu tiên và nó tương ứng với chữ cái thứ i của bảng chữ cái thứ hai, được viết trong trường $\text{transf}[i].\text{st2}$. Khi thực hiện phép dịch, trong mảng $\text{int translate}[]$, chúng ta chỉ viết số của các chữ cái đã dịch và nếu chúng ta đi đến quyết định, chúng ta in các chữ cái của bảng chữ cái thứ hai tương ứng với các số này. Trong chương trình, chúng ta đã sử dụng bảng chữ cái tiếng Latinh và các số từ 0 đến 9 làm dữ liệu đầu vào mẫu. Bảng chữ cái thứ hai bao gồm mã Morse tương ứng cho mỗi ký hiệu.

Chương trình 6.8. Các bản dịch mật mã (608translat.c)

```
#include <stdio.h>
#include <string.h>
#define MAXN 40 /* Số lớn nhất tương ứng giữa các chữ cái */
#define MAXTL 200 /* Độ dài lớn nhất của từ để dịch*/
/* Số lượng tương ứng*/
const unsigned n = 38;
/* Từ phải dịch */
char *str1 = "101001010";

struct transType {
    char *st1;
    char *st2;
};
struct transType transf[MAXN];

unsigned translation[MAXTL], pN, total = 0;
```

* Ví dụ dùng bảng morse: 0 là tạch, còn 1 là tè */

```
void initLanguage(void)
{   transf[0].st1 = "A"; transf[0].st2 = "01";
    transf[1].st1 = "B"; transf[1].st2 = "1000";
    transf[2].st1 = "C"; transf[2].st2 = "011";
    transf[3].st1 = "D"; transf[3].st2 = "110";
    transf[4].st1 = "E"; transf[4].st2 = "100";
    transf[5].st1 = "F"; transf[5].st2 = "0";
    transf[6].st1 = "G"; transf[6].st2 = "0001";
    transf[7].st1 = "H"; transf[7].st2 = "1100";
    transf[8].st1 = "I"; transf[8].st2 = "00";
    transf[9].st1 = "K"; transf[9].st2 = "0111";
    transf[10].st1 = "L"; transf[10].st2 = "101";
    transf[11].st1 = "M"; transf[11].st2 = "0100";
    transf[12].st1 = "N"; transf[12].st2 = "11";
    transf[13].st1 = "O"; transf[13].st2 = "10";
    transf[14].st1 = "P"; transf[14].st2 = "111";
    transf[15].st1 = "Q"; transf[15].st2 = "0110";
    transf[16].st1 = "R"; transf[16].st2 = "010";
    transf[17].st1 = "S"; transf[17].st2 = "000";
    transf[18].st1 = "T"; transf[18].st2 = "1";
    transf[19].st1 = "U"; transf[19].st2 = "001";
    transf[20].st1 = "V"; transf[20].st2 = "0010";
    transf[21].st1 = "X"; transf[21].st2 = "0000";
    transf[22].st1 = "Y"; transf[22].st2 = "1010";
    transf[23].st1 = "Z"; transf[23].st2 = "1110";
    transf[24].st1 = "W"; transf[24].st2 = "1111";
    transf[25].st1 = "Â"; transf[25].st2 = "1101";
    transf[26].st1 = "Ă"; transf[26].st2 = "0011";
    transf[27].st1 = "Ê"; transf[27].st2 = "0101";
    transf[28].st1 = "1"; transf[28].st2 = "01111";
    transf[29].st1 = "2"; transf[29].st2 = "00111";
    transf[30].st1 = "3"; transf[30].st2 = "00011";
    transf[31].st1 = "4"; transf[31].st2 = "00001";
    transf[32].st1 = "5"; transf[32].st2 = "00000";
    transf[33].st1 = "6"; transf[33].st2 = "10000";
    transf[34].st1 = "7"; transf[34].st2 = "11000";
    transf[35].st1 = "8"; transf[35].st2 = "11100";
    transf[36].st1 = "9"; transf[36].st2 = "11110";
    transf[37].st1 = "0"; transf[37].st2 = "11111";
}
```

```

/* In bản dịch */
void printTranslation(void)
{ unsigned i;
    total++;
    for (i = 0; i < pN; i++)
        printf("%s", transf[translation[i]].st1);
    printf("\n");
}

/* tìm chữ cái tiếp sau */
void nextLetter(unsigned count)
{ unsigned i, k;
    if (count == strlen(str1)) { printTranslation(); return; }
    for (k = 0; k < n; k++) {
        unsigned len = strlen(transf[k].st2);
        for (i = 0; i < len; i++)
            if (str1[i + count] != transf[k].st2[i]) break;
        if (i == len) {
            translation[pN++] = k;
            nextLetter(count + strlen(transf[k].st2));
            pN--;
        }
    }
}

int main()
{
    printf("Danh sách tất cả khả năng dịch: \n");
    initLanguage();
    pN = 0;
    nextLetter(0);
    printf("Số lượng chung cách dịch khác nhau: %u \n", total);
    return 0;
}

```

Bài tập

- 6.34. Sửa đổi chương trình trên để tìm bản dịch có độ dài tối thiểu. Những điều kiện nào để bỏ cây đối với các ứng cử viên cho quyết định có thể được áp dụng?

6.4. Phương pháp nhánh và ranh giới

Một trường hợp đặc biệt của phương pháp tìm kiếm quay lui là phương pháp các nhánh và các đường biên. Nó áp dụng cho các bài toán áp dụng hai điều kiện sau:

1) Mỗi ứng cử viên cho một quyết định được so sánh giá trị - *giá cả* và mục tiêu là tìm ra *giải pháp tối ưu* (ví dụ: giải pháp có giá tối thiểu).

2) Mỗi giải pháp của bài toán phải có thể được trình bày dưới dạng thu được từ các phần mở rộng liên tiếp của các giải pháp từng phần với giá tăng đơn điệu.

Trong Chương 5 (xem 5.4.4.) Chúng ta đã giải quyết vấn đề của khách du lịch thương mại bằng phương pháp phân nhánh và ranh giới. Trong bài toán này, cả hai điều kiện đã được đáp ứng:

1) Mỗi lần từ đồ thị (và cụ thể là chu trình Hamilton), một giá được so sánh - tổng trọng lượng của các sườn mà nó chứa. Chúng ta đang tìm kiếm một chu kỳ Hamilton với giá tối thiểu.

2) Vì chúng ta đã đưa ra hạn chế bổ sung là trọng số của các cạnh của đồ thị phải là số dương nên điều kiện thứ hai cũng được đáp ứng: Giá của mỗi nghiệm riêng trong đó các đỉnh tham gia (v_1, v_2, \dots, v_k) nhỏ hơn giá của phần mở rộng tùy ý của nó $(v_1, v_2, \dots, v_k, v_{k+1})$, thu được bằng cách thêm một đỉnh khác.

Điều quan trọng đối với phương pháp nhánh và đường biên là cây đê quy mà người ứng viên xin quyết định thu được có thể bị "cắt" trên cơ sở một nguyên tắc khác: Nếu một giải pháp từng phần có giá lớn hơn hoặc bằng giá trị nhỏ nhất được tìm thấy vào thời điểm hiện tại, không có ý nghĩa gì khi mở rộng thêm, vì điều này chắc chắn sẽ không dẫn đến một giải pháp tốt hơn. Chúng ta đã thực hiện một phần tương tự về những quyết định vô vọng trong bài toán dành cho khách du lịch thương mại. Chúng ta sẽ xem xét một bài toán NP-đầy đủ khác và biến thể của nó, trong đó có thể tìm lời giải bằng phương pháp nhánh và đường biên.

6.4.1. Bài toán ba lô (lựa chọn tối ưu)

Bài toán 1. Cho một ba lô có khối lượng (độ chứa) M kg. Cho N vật, mỗi vật nặng m_i ($1 \leq i \leq N$). Kiểm tra xem có một tập hợp con các đồ vật lắp đầy ba lô chính xác không, tức là tổng trọng lượng của chúng chính xác là M .

Bài toán 2. (phiên bản tối ưu hóa tổng quát của bài trước) Một ba lô có thể chứa M kg được đưa ra. Cho trước N vật, mỗi vật có khối lượng m_i và giá trị (giá) c_i . Tìm một tập hợp con các món có tổng chi phí lớn nhất để đựng vừa ba lô, tức là tổng khối lượng của chúng nhỏ hơn hoặc bằng M .

Trong một số trường hợp đặc biệt, ví dụ khi trọng số là số nguyên và có đủ bộ nhớ ($\Theta(M.N)$), các thuật toán hiệu quả có thể được áp dụng để giải các bài toán trên dựa trên nguyên tắc tối ưu động (Chương 8). Tuy nhiên, trong trường hợp tổng quát, khi giá và trọng số là số thực, hai bài toán NP-đầy đủ này được giải quyết triệt để. Chúng ta sẽ xem xét thứ hai trong số họ và giải quyết nó bằng phương pháp các nhánh và ranh giới.

Chúng ta sẽ tạo ra tất cả các giải pháp có thể bằng cách tìm kiếm có trả lại và trong quá trình tạo, chúng ta sẽ giữ lại giải pháp tối ưu. Trong bài toán thứ hai, một tập hợp các đối tượng được đưa ra. Từ tất cả 2^N tập con của nó, ta phải chọn một tập thỏa mãn điều kiện của bài toán: tổng trọng lượng của các vật trong đó không được vượt quá M và tổng giá trị của chúng phải lớn nhất.

Chúng ta hãy xem xét những điểm chính trong việc lập thuật toán. (Để đơn giản hóa ký hiệu, chúng ta sẽ xem c_i và m_i là các mảng $c[i]$ và $m[i]$, tương ứng.)

Ta có một nghiệm riêng (tập hợp con của các đối tượng) trong đó các đối tượng $A = a_1, a_2, \dots, a_k$ được lấy. Theo $S(a_1, a_2, \dots, a_k)$ chúng ta có nghĩa là tổng $S(a_1, a_2, \dots, a_k) = m[a_1] + m[a_2] + \dots + m[a_k]$ bằng trọng lượng của chúng. Đối với tập hợp con A , chúng ta cố gắng thêm một đối tượng mới a_{k+1} trong số những đối tượng chưa được lấy cho đến nay:

- Nếu $S(a_1, a_2, \dots, a_k) + m[a_{k+1}] \leq M$ thì sau đó phép cộng đối tượng là có thể và ta tiếp tục khai triển đệ quy tập con A theo mọi cách có thể. .

- Nếu không tồn tại đối tượng a_{k+1} sao cho $S(a_1, a_2, \dots, a_k) + m[a_{k+1}] \leq M$ thì theo đó tập a_1, a_2, \dots, a_k là một giới hạn (tức là không thể mở rộng thêm nữa).

Khi đó giá trị của nó $c[A] = c[a_1] + c[a_2] + \dots + c[a_k]$ được tính và nếu nó lớn hơn giá trị lớn nhất tìm được cho đến nay, nó được giữ nguyên:

Sơ đồ chọn đồ vật

```
void generate(i) {
    if (<nếu trọng lượng chung của vật đã chọn > M>) return;
    if (i == N) {
        /* trường hợp cận biên, kiểm tra có giải pháp nào tốt hơn*/
        /* trong số tối đa được tìm thấy cho đến nay */
        return;
    }
    <lấy vật thứ i>;
    generate(i+1);
    <loại bỏ vật thứ i>;
    generate(i+1); /* nghĩa là chọn vật thứ i được */
}
```

Trong thực tế, chúng ta đã xác định khi nào một giải pháp từng phần có thể được mở rộng. Vì vậy, chúng ta đảm bảo rằng không phải tất cả 2 tập con N -có thể được xem xét, mà chỉ những tập mà tổng trọng lượng của các đối tượng nhỏ hơn hoặc bằng M . Tuy nhiên, điều này không kết thúc vấn đề cắt cây đệ quy của các ứng cử viên.

Trong trường hợp của bài toán người bán hàng, nơi tìm kiếm một chu trình Hamilton tối thiểu, một giải pháp từng phần được tuyên bố là vô vọng nếu chiều dài của con đường đang được xây dựng trở nên lớn hơn mức tối thiểu được tìm thấy cho đến nay. Khi tìm giải pháp tối đa, như trong bài toán chọn tối ưu, kiểu cắt này phải được thay đổi: Trong quá trình xây dựng một tập hợp con các đối tượng (như trong quy trình giản đồ trên) chúng ta đã lấy các đối tượng $A = \{a_1, a_2, \dots, a_k\}$, và chúng ta đã bỏ qua các đối tượng $B = \{b_1, b_2, \dots, b_p\}$. Với VT , chúng ta sẽ biểu thị tổng các giá trị của tất cả các đối tượng, và với $Vmax$, chúng ta sẽ biểu thị giá trị lớn nhất thu được từ các giải pháp đã thử nghiệm cho đến nay. Sau đó

nếu

$$VT - c[b_1] - c[b_2] - \dots - c[b_p] \leq Vmax,$$

theo đó là không có ích lợi gì khi mở rộng giải pháp hiện tại thêm nữa. Điều này là do ngay cả khi có thể bao gồm tất cả các mục khác, giá trị tối đa mà chúng ta sẽ nhận được là:

$$\begin{aligned} & (\text{giá trị của các mục đã lấy cho đến nay}) + (\text{giá trị còn lại}) = \\ &= (c[a1] + \dots + c[ak]) + (VT - (c[a1] + \dots + c[ak] + c[b1] + \dots + c[bp])) = \\ &= VT - c[b1] - c[b2] - \dots - c[bp], \end{aligned}$$

trong khi đó, nếu biểu thức sau nhỏ hơn $Vmax$, thì giải pháp hiện tại là vô vọng và việc xem xét thêm sẽ không dẫn đến kết quả tốt hơn. Sau đây là một lược đồ chi tiết hơn của hàm đệ quy `generate()`:

Hàm đệ quy `generate()`

```
float Ttemp; /* Trọng số chung của các vật đã chọn*/
float Vtemp; /* Giá chung của các vật đã chọn*/
float VmaX; /* Giá cực đại của lời giải được chọn cho đến nay*/
float totalV; /* Giá chung của vật còn lại*/

void generate(unsigned i)
{ if (Ttemp > K) return;
  if (Vtemp + totalV < VmaX) return; /* giải pháp vô vọng */
  if (i == N) {
    if (Vtemp > VmaX) {
      <lưu lời giải>;
      VmaX = Vtemp;
    }
    return;
  }
  Vtemp += c[i]; Ttemp += m[i]; totalV -= c[i]; /*lấy vật i */
  generate(i+1);
  Vtemp -= c[i]; Ttemp -= m[i]; /* loại vật i */
  generate(i+1); /* tức là khi lấy đổi tượng thứ i bị "bỏ qua" */
  totalV += c[i];
}

int main()
{
  totalV = c[1] + c[2] + ... + c[N];
```

```

    generate(0);
    return 0;
}

```

Để có được một chương trình làm thật, chúng ta cũng phải chỉ rõ cách chúng ta sẽ thực hiện việc đưa vào và loại trừ chủ thể thứ i . Với mục đích này, chúng ta sẽ nhập $token[]$ và một biến tN , hiển thị số phần tử của nó. Chúng ta sẽ lưu giải pháp tốt nhất được tìm thấy cho đến nay trong mảng $saveTaken[]$, với phần tử sn . Khi kết thúc hàm $main()$, chúng ta sẽ in ra giải pháp tối ưu tìm được. Dữ liệu đầu vào là N , M , $c[N]$ và $m[N]$ được đặt là hằng số ở đầu chương trình.

Chương trình 6.9. Bài toán ba lô (609bagrec.c)

```

#include <stdio.h>
#define MAXN 100

const unsigned n = 10;
const double M = 10.5;
const double c[MAXN] = {10.3,9.0,12.0,8.0,4.0,8.4,9.1,17.0,6.0,9.7};
const double m[MAXN] = {4.0,2.6,3.0,5.3,6.4,2.0,4.0,5.1,3.0,4.0};

unsigned taken[MAXN], saveTaken[MAXN], tn, sn;
double VmaX, Vtemp, Ttemp, totalV;

void generate(unsigned i)
{ unsigned k;
  if (Ttemp > M) return;
  if (Vtemp + totalV < VmaX) return;
  if (i == n) {
    if (Vtemp > VmaX) /* Giữ nghiệm tối ưu */
      VmaX = Vtemp; sn = tn;
    for (k = 0; k < tn; k++)
      saveTaken[k] = taken[k];
  }
  return;
}

taken[tn++] = i; Vtemp += c[i]; totalV -= c[i]; Ttemp += m[i];
generate(i + 1);

```

```

tn--; Vtemp -= c[i]; Ttemp -= m[i];
generate(i + 1);
totalV += c[i];
}

int main() {
    unsigned i;
    tn = 0; VmaX = 0; totalV = 0;
    for (i = 0; i < n; i++)
        totalV += c[i];
    generate(0);
    printf("Giá lớn nhất: %.2lf\nChọn vật: \n", VmaX);
    for (i = 0; i < sn; i++)
        printf("%u ", saveTaken[i] + 1);
    printf("\n");
    return 0;
}

```

Kết quả thực hiện chương trình:

Giá tối đa: 37,40

Các mục đã chọn:

3 6 8

Bài tập

- ▷ 6.35. Hãy biên dịch và thực hiện một bài toán giải thuật toán 1.
- ▷ 6.36. Sửa đổi chương trình trên để tìm tất cả các giải pháp với giá tối đa.

6.5. Các chiến lược tối ưu cho trò chơi

Ví dụ về cờ vua, mà chúng ta đã đưa ra ở đầu chương, chúng ta sẽ xem xét thêm hai lần nữa - ở đây và trong 6.5.2. Trong 6.3.4 chúng ta đã chỉ ra một cách khá sơ đồ về cách một cái cây có thể được xây dựng dựa trên tất cả các loại chuyển động. Nay giờ, khi nhìn, chúng ta sẽ không sử dụng một cái cây, mà là một đồ thị có định hướng. Mỗi cấu hình cờ vua có thể sẽ được biểu diễn bằng một đỉnh của đồ thị $G(V, E)$. Cạnh $(i, j) \in E$ khi và chỉ khi của cấu hình tương ứng

với i , cấu hình j có thể nhận được bằng cách thực hiện một bước di chuyển. Chúng ta sẽ xác định một số khái niệm:

- Cấu hình cờ vua được gọi là thiết bị đầu cuối nếu trò chơi kết thúc (quân đen, quân trắng hoặc hòa). Trong cờ vua, một trận hòa xảy ra khi:
 - *nam* - không thể có nước đi nào trước người chơi đang đến lượt chơi.
 - *bị chiếu vĩnh viễn* - lặp lại cùng một nước đi ba lần (thường bị ép bằng cờ vua của một trong các đối thủ, người đang tìm kiếm một kết quả hòa).
 - *không có khả năng chiến thắng* của một trong hai đối thủ (có quá ít và / hoặc quân cờ yếu / không thể cơ động trên bàn cờ).
 - *thỏa thuận hòa* giữa các kỳ thủ (kể cả trường hợp “chủ quan” này cũng không nên bỏ qua và có giá trị trong máy tính nhận thức cờ vua).

Khi có cấu hình đầu cuối, đindh tương ứng của đồ thị không có người thừa kế.

- Cấu hình không đầu cuối được gọi là *chiến thắng* nếu có ít nhất một người kế nhiệm là người thua cuộc (tức là chỉ với một nước đi ta có thể đạt đến một cấu hình mà đối thủ đang di chuyển và mỗi nước đi đều dẫn đến một trận thua trong tương lai).
- Cấu hình không đầu cuối được gọi là *thua* nếu tất cả các cấu hình kế tiếp của nó đều là cấu hình thắng (tức là chúng ta chơi nước đi nào thì đối thủ của chúng ta sẽ có nước đi thắng ngay sau đó).
- Bất kỳ cấu hình nào khác là *không chắc chắn* (nếu cả hai người chơi chơi tối ưu, trò chơi sẽ kết thúc với tỷ số hòa).

Vì vậy, mục tiêu của một chương trình chơi cờ hoàn hảo là luôn cố gắng đạt được cấu hình chiến thắng. Nhất cử nhát động, chiến thắng chỉ là vấn đề thời gian và hoàn toàn chắc chắn, bất kể đối thủ ra sao.

6.5.1. Trò chơi "X" và "O"

Xây dựng một biểu đồ đại diện cho một phân tích đầy đủ của một trò chơi cờ vua không phải là thế mạnh của kỹ thuật hiện tại (nó không cho phép bảo tồn hoặc nghiên cứu một biểu đồ có khoảng 10.100 đỉnh). Do đó, chúng ta sẽ giải quyết vấn đề về chiến lược chiến thắng với một trò chơi đơn giản hơn nhưng cũng nổi tiếng:

Hai người chơi trên bảng có kích thước 3×3 . Ở mỗi nước đi, người chơi lần lượt điền vào các ô trên bảng - người chơi 1 sử dụng ký hiệu "X" và người chơi 2 sử dụng ký hiệu "O". Người chiến thắng là người đầu tiên điền vào toàn bộ hàng ngang, hàng dọc hoặc một trong hai đường chéo chính của bàn cờ.

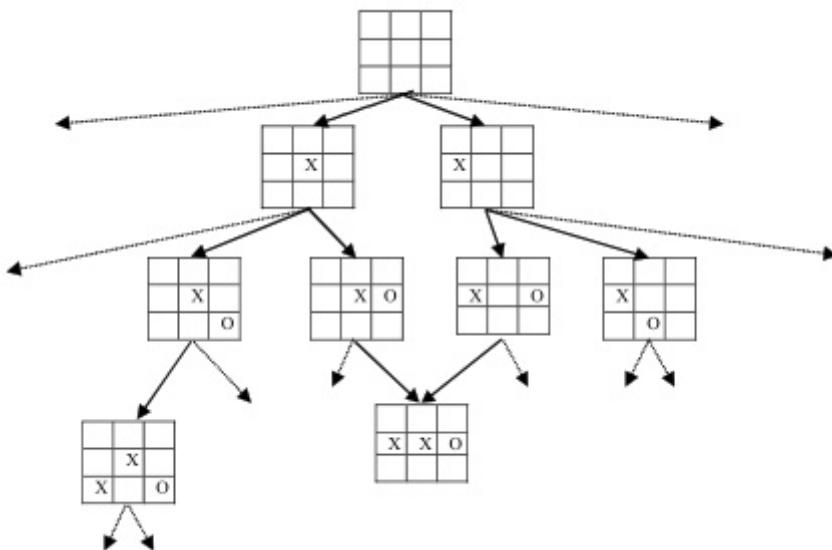
Chúng ta sẽ trình bày các cấu hình có thể có trong trò chơi dưới dạng các nút đồ thị (lưu ý rằng, không giống như cờ vua, trong trò chơi "X" và "O", đồ thị là dòng xoay). Một phần của đồ thị được thể hiện trong Hình 6.10.

Các điều khoản của đoạn trước cũng có giá trị ở đây - ví dụ trong Hình 6.11, cấu hình T_3 và T_4 là đầu cuối (người chơi 1 đã thắng trò chơi), cấu hình T_2 là thua (bất kỳ nước đi nào mà người chơi 2 chơi, anh ta sẽ thua), và T_1 là chiến thắng.

Chúng ta sẽ đưa ra một thuật toán có thể chơi trò chơi "X" và "O" một cách tối ưu, thực hiện hết hoàn toàn (tất cả các cấu hình có thể nhận được từ board trong một lần):

Sơ đồ kiểm tra vị trí chơi

```
/*
 * Trả lại: 1 - nếu cấu hình mang lại lợi thế cho người chơi,
 *           2 - nếu nó là một thế thua cuộc
 *           3 - nếu nó không xác định
 */
int checkPosition(int player, board)
{
    if (<điểm là thiết bị đầu cuối>) {
        if (<trò chơi kết thúc>) return 3; /* cờ hòa */
        if (<player == người chơi thắng>) return 1;
        if (<player != người chơi thắng>) return 2;
    } else {
```



Hình 6.10. Đồ thị trò chơi "X" và "O"

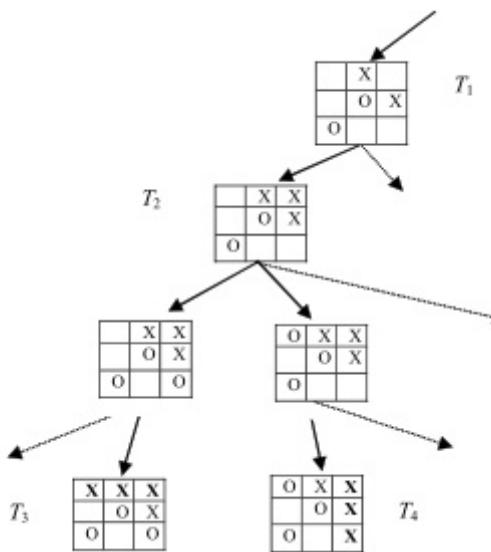
```

if (checkPosition(<người chơi khác>,boardi)==1 với mọi i=1,..,n)
  return 2;
if (checkPosition(<người chơi khác>,boardi)==2 với mọi i=1,..,n)
  return 1;
return 3;
}
}
  
```

Sơ đồ trên sử dụng cùng một giá trị để chỉ ra một vị trí không xác định và một vị trí cuối cùng mà tại đó trò chơi kết thúc với tỷ số hòa. Như đã lưu ý trong đoạn 6.5, thực tế là một vị trí không chắc chắn (ngay cả khi cây trò chơi được xem xét toàn bộ) có nghĩa là với cách chơi tối ưu cho cả hai người chơi, trò chơi chắc chắn sẽ kết thúc với tỷ số hòa.

Lược đồ của hàm checkPosition() là phổ quát. Nó có thể được áp dụng trong tất cả các trò chơi liên quan đến hai người chơi luôn phiên và mỗi cấu hình có thể nhận được trong một lần di chuyển từ người khác.

Nó cũng có thể được sử dụng trong cờ vua, nhưng do kích thước của cây, nó sẽ chỉ giúp ích trong một số trường hợp đơn giản hơn



Hình 6.11. Các vị trí kết thúc trong cây trò chơi



Hình 6.12. Chiếu tướng sau 3 nước đi (trắng đi trước)

(ví dụ, Hình 6.12 cho thấy một vị trí chiến thắng cho quân trắng). Với nó, chúng ta có thể giải quyết các vấn đề cờ vua cổ điển kiểu "cờ tướng trong 3 nước đi" và các bài toán khác.

Sau đây là cách triển khai có thể có của hàm checkPosition() cho các trò chơi "X" và "O". Chúng ta sử dụng hàm terminal() để kiểm tra xem cấu hình có phải là thiết bị đầu cuối hay không và nếu

có, trả về người chiến thắng là ai hoặc trò chơi đã kết thúc với tỷ số hòa:

Chương trình 6.10. Trò chơi tic và tac (610tictac.c)

```
#include <stdio.h>
/* Người chơi bắt đầu */
const char startPlayer = 2;
/* Cấu hình bắt đầu */
char board[3][3] = {
    { '.', '.', '.' },
    { '.', 'X', '.' },
    { 'X', '.', 'O' }};

/* kết quả: 1, nếu cấu hình kết thúc và chiến thắng là người 1,
 * 2, nếu cấu hình kết thúc và chiến thắng là người 2,
 * 3, nếu cấu hình kết thúc và cờ hòa
 * 0, nếu cấu hình không kết thúc
 */
char terminal(char a[3][3])
{ unsigned i, j;
    for (i = 0; i < 3; i++) {
        /* kiểm tra các đường ngang */
        for (j = 0; j < 3; j++)
            if (a[i][j] != a[i][0]) break;
        if (3 == j && a[i][0] != '.') {
            if (a[i][0] == 'X') return 1; else return 2;
        }

        /* kiểm tra các đường đứng */
        for (j = 0; j < 3; j++)
            if (a[j][i] != a[0][i]) break;
        if (3 == j && a[i][0] != '.') {
            if (a[0][i] == 'X') return 1; else return 2;
        }

        /* kiểm tra các đường chéo */
        if (a[0][0] == a[1][1] && a[1][1] == a[2][2] && a[1][1] != '.')
            if (a[0][0] == 'X') return 1; else return 2;
        if (a[2][0] == a[1][1] && a[1][1] == a[0][2] && a[1][1] != '.')
            if (a[2][0] == 'X') return 1; else return 2;
    }
}
```

```

/* phải chăng là hòa (phải chăng các vị trí đều bằng) */
for (i = 0; i < 3; i++)
    for (j = 0; j < 3; j++)
        if (a[i][j] == '.') return 0;
return 3;
}

/* trả về: 1, nếu cấu hình không thắng cho người chơi,
 * 2, nếu là thua và
 * 3, nếu không xác định */
char checkPosition(char player, char board[3][3])
{ unsigned i, j, result;
int t = terminal(board);
if (t) {
    if (player == t) return 1;
    if (3 == t) return 3;
    if (player != t) return 2;
}
else {
    char otherPlayer, playerSign;
    if (player == 1) { playerSign = 'X'; otherPlayer = 2; }
    else { playerSign = 'O'; otherPlayer = 1; }
/* char board[3][3];
* xác định vị trí
*/
for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
        if (board[i][j] == '.') {
            board[i][j] = playerSign;
            result = checkPosition(otherPlayer, board);
            board[i][j] = '.';
            if (result == 2) return 1; /* cấu hình thua cho đối thủ, */
            /* suy ra thắng cho người chơi */
            if (result == 3) return 3; /* cấu hình không xác định */
        }
    }
}
/* tất cả bước sau đều cấu hình thắng và
* suy ra nó thua cho người chơi */

```

```

        return 2;
    }
}

int main() {
    printf("%u\n", checkPosition(startPlayer, board));
    return 0;
}

```

Trong vài điểm tiếp theo chúng ta sẽ xem xét vấn đề này - để so sánh “chất lượng” của hai vị trí không xác định, cũng như việc chặt một phần cây sẽ được nghiên cứu.

Bài tập

▷ 6.37. Hãy viết một chương trình mô phỏng trò chơi "X" -chọn và "O" giữa người và máy tính, và máy tính phải chơi một cách tối ưu: để giành chiến thắng bất cứ khi nào anh ta có được cơ hội như vậy, tức là trong trường hợp lối của con người, và không mắc lối cho mình. Khi vị trí của máy tính không chắc chắn, nó có thể thực hiện bất kỳ hành động nào không gây tổn thất.

▷ 6.38. Hãy viết chương trình chơi chữ "X" và "O" trong trường hợp chung, khi một bảng có kích thước $n \times n$ được đưa ra, và người chiến thắng là người chơi tạo được một hàng, cột hoặc đường chéo của m ký tự. Trong biến thể phổ biến nhất, m là 5.

▷ 6.39. Hãy chứng minh rằng với bàn là 3×3 , nếu cả hai người chơi đều chơi đúng, trò chơi phải kết thúc với tỷ số hòa.

▷ 6.40. Tìm số của:

- tất cả các loại cấu hình có thể nhận được trong trò chơi
- tất cả các loại cấu trúc gỗ (so sánh với a))
- số lượng lá trên cây (tức là cấu hình đầu cuối)

6.5.2. Nguyên tắc minimum và maximum

Chúng ta sẽ tiếp tục với trò chơi "X" và "O". Không giống như đoạn trước, bây giờ chúng ta sẽ trình bày các chuyển động có thể xảy ra không phải với đồ thi (như trong Hình 6.10), Mà là với một

cây, tại mỗi đỉnh có một số ước lượng được so sánh. Nó xác định vị trí tối ưu như thế nào đối với người chơi 1. Chúng ta sẽ bắt đầu với danh sách trên cây (cấu hình thiết bị đầu cuối): một trang tính có xếp hạng $+\infty$ nếu người chơi 1 thắng, xếp hạng $-\infty$ nếu người chơi 2 thắng và 0 xếp hạng nếu 0 trận đấu kết thúc với tỷ số hòa. Xếp hạng của tất cả các đỉnh khác được xác định bởi những người kế thừa của chúng như sau: Đối với đỉnh i với những người thừa kế i_1, i_2, \dots, i_k , xếp hạng $V[i]$ được xác định dựa trên *nguyên tắc minimum và maximum*:

$$V[i] = \begin{cases} \max(V[i_1], V[i_2], \dots, V[i_k],) & \text{nếu bước đi là người chơi 1} \\ \min(V[i_1], V[i_2], \dots, V[i_k],) & \text{nếu bước đi là người chơi 2} \end{cases}$$

Một đỉnh được gọi là *đỉnh tối đa hóa* nếu người chơi 1 đang tiến hành cấu hình phù hợp với nó, nếu không nó được gọi là *đỉnh tối thiểu* (các định nghĩa này dựa trên thực tế là xếp hạng đỉnh được xác định là mức tối đa / tối thiểu của xếp hạng của người kế nhiệm). Sau đây là một hàm (dựa trên phương pháp tìm kiếm trả về) tính toán ước lượng của bất kỳ đỉnh nào từ cây.

Hàm đánh giá trên mỗi đỉnh của cây

```
value minimax(<đỉnh i>) {
    if (<i là lá>) return <đánh giá lá>;
    <Gọi (i1 , i2 ,..., in ) là những thừa kế của i>;
    if (<i là đỉnh với nhỏ nhất>
        return min(minimax(i1 ), ..., minimax(in ));
    if (<i là đỉnh với cực đại hóa>
        return max(minimax(i1 ), ..., minimax(in ));
}
```

Vì vậy, bài toán của việc thực hiện tốt nhất tiếp theo là tìm ra người kế vị hàng đầu với số điểm cao nhất. Trong trò chơi "X" -chess và "O" điểm của các đỉnh-lá có thể là $-\infty, +\infty$ và 0. Do đó điểm của mỗi đỉnh của cây nhất thiết phải là một trong ba giá trị này. Tuy nhiên, trong nhiều trò chơi khác, có thể các ước lượng khác nhau cho các lá nhiều hơn hoặc việc đánh giá phần ngọn của cây được xác định mà không cần phải kiểm tra toàn bộ cây con của nó (tức là, đánh giá được thực hiện bởi một số tiêu chí chủ quan - xem 6.5.4).

Đặc điểm của hàm `minimax()` là nó kiểm tra từng đỉnh của cây. Thường thì điều này là không cần thiết.

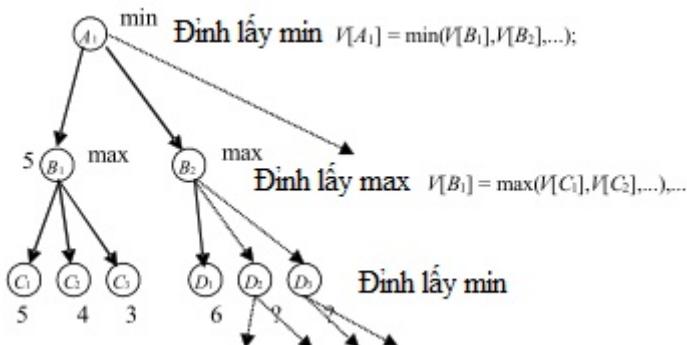
Bài tập

- 6.41. Viết chương trình mô phỏng trò chơi chữ "X" và chữ "O" giữa người và máy tính, thực hiện nguyên tắc tối thiểu và tối đa.

6.5.3. Nhát cắt alpha-beta

Nhát cắt alpha-beta là một phương pháp giảm các đỉnh được nghiên cứu trên nguyên tắc tối thiểu và tối đa. Hãy xem hai ví dụ.

Trong Hình 6.13 một cây được hiển thị trong đó thuật toán phải tính toán ước lượng đỉnh để tối thiểu hóa A_1 . Điểm của A_1 là điểm tối thiểu trong số các điểm kế thừa của anh ta $V[B_1], V[B_2], V[B_3]$, v.v.

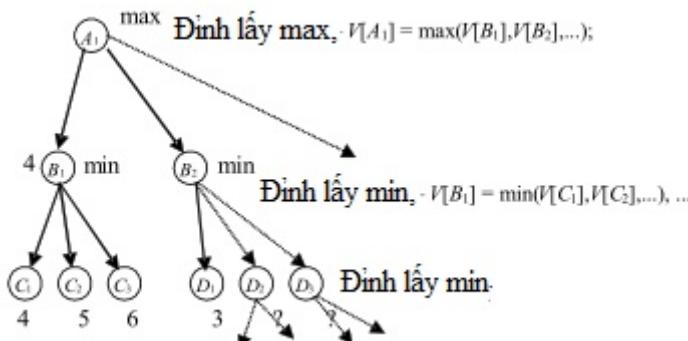


Hình 6.13. Nhát cắt beta

Thuật toán đầu tiên cố gắng tìm giá trị của ước lượng $V[B_1]$ - nó được xác định là giá trị lớn nhất trong số các ước lượng kế tiếp của nó. Giả sử rằng các ước lượng kế tiếp của B_1 đã được tính toán, vì vậy $V[B_1] = \max(V[C_1, C_2, C_3]) = \max(5, 4, 3) = 5$. Vì như A_1 là đỉnh cực tiểu, từ $V[B_1] = 5$ ước tính của A_1 sẽ nhỏ hơn hoặc bằng 5. Quá trình thu thập thông tin tiếp tục với đỉnh B_2 . Giá trị $V[B_2]$ là giá trị lớn nhất trong các giá trị kế tiếp của nó là D_1, D_2, D_3, \dots . Chúng ta đến với phần thực của việc nhát cắt alpha-beta - trong ví dụ này, chúng ta giả định rằng ước lượng thu được sau khi nghiên cứu cây

con có gốc D_1 là $V[D_1] = 6$. Theo đó ước lượng của B_2 sẽ ít nhất là 6 (chúng ta đang tìm kiếm giá trị tối đa). Mặt khác, chúng ta đã thấy rằng $V[A_1] \leq 5$. chúng ta không quan tâm và sẽ không được nghiên cứu, vì chúng ta đã có được rằng $V[B_2] \geq 6$ và $V[A_1] \leq 5$.

Nhát cắt alpha được áp dụng khi xem xét một đỉnh tối đa hóa. Xét ví dụ trong Hình 6.14: Giả sử chúng ta đã tính được $V[C_1] = 4, V[C_2] = 5, V[C_3] = 6$. Như vậy $V[B_1] = \min(V[C_1], V[C_2], V[C_3]) = \min(4, 5, 6) = 4$. Theo đó $V[A_1] \geq 4$. Tiếp theo, hóa ra $V[D_1] = 3$, theo đó là $V[B_2] \leq 3$. Từ $V[A_1] \geq 4$ và $V[B_2] \leq 3$, việc nghiên cứu thêm về những người thừa kế của B_2 là không cần thiết (nhát cắt alpha).



Hình 6.14. Nhát cắt alpha

Việc triển khai nhát cắt alpha và beta được thực hiện như sau: Đối với mỗi đỉnh được duyệt, ngoài đánh giá của nó, hai giá trị khác được tính toán - *giá trị alpha* và *giá trị beta* của đỉnh:

Giá trị alpha của đỉnh

Ta khởi tạo các giá trị alpha như sau: Nếu đỉnh là lá thì giá trị alpha là $-\infty$, ngược lại thì trùng với ước lượng của nó. Sau đó, khi thu thập thông tin, giá trị alpha của mỗi đỉnh *tối đa hóa* được xác định là giá trị lớn nhất của các ước tính của các giá trị kế thừa được nghiên cứu cho đến nay. Đối với các đỉnh cực tiểu, giá trị alpha là giá trị alpha của *giá trị đỉnh trước đó*.

Giá trị beta các đỉnh

Các giá trị beta của đỉnh lá tính được khởi tạo với các ước tính

của chúng. Tất cả các đỉnh khác được khởi tạo với giá trị beta là $+\infty$. Khi duyệt sau đó, giá trị beta của đỉnh tối thiểu hóa được xác định là giá trị nhỏ nhất của các ước tính của các giá trị kế thừa được nghiên cứu cho đến nay. Giá trị beta của đỉnh tối đa hóa là giá trị beta của giá trị tiền nhiệm.

Bài toán này đảm bảo rằng:

- Điểm đỉnh sẽ luôn không nhỏ hơn giá trị alpha của đỉnh và không lớn hơn giá trị beta của nó.
- Giá trị alpha và beta có thể thay đổi khi bạn duyệt qua, nhưng chắc chắn rằng giá trị alpha sẽ không bao giờ giảm và giá trị beta sẽ tăng lên.
- Giá trị alpha của mỗi đỉnh (trừ đỉnh gốc) luôn lớn hơn hoặc bằng giá trị alpha của đỉnh trước của nó.
- Giá trị beta của mỗi đỉnh (trừ đỉnh gốc) luôn nhỏ hơn hoặc bằng giá trị beta của đỉnh trước của nó.

Chúng ta sẽ hiển thị một sơ đồ mà cây có thể được duyệt. Hàm `minimaxCutoff()` được gọi với các tham số đỉnh gốc cây, $-\infty$ và $+\infty$ (giá trị alpha và beta, tương ứng):

Nhát cắt minimax

```
value minimaxCutoff (N, A, B) {
    Na = A; /* Na là giá trị alpha của đỉnh được tính */
    Nb = B; /* Nb là giá trị beta của đỉnh sẽ được tính */
    if (<N là đỉnh lá>) return <giá trị giá của lá>;
    if (<N là đỉnh lấy min>
        for (<với mọi thù kế Ni của N>)
            val = minimaxCutoff (Ni, Na, Nb);
            Nb = min (Nb, val);
            if (Nb <= Na) break; /* nhát cắt alpha-beta */
        }
        return Nb;
    }
    else /* * nếu N là đỉnh lấy max*/
        for (<với mỗi đỉnh thù kế Ni của N>)
            val = minimaxCutoff (Ni, Na, Nb);
            Na = max (Na, val);
            if (Na >= Nb) break; /* nhát cắt alpha-beta */
    }
}
```

```

    } return Na;
}

```

Trong hai dòng (có chữ " nhát cắt alpha-beta "), nghiên cứu thêm về alpha-beta bị gián đoạn, vì nó là vô vọng.

Bài tập

- ▷ 6.42. Viết chương trình mô phỏng trò chơi "X-lần" và "O" giữa người và máy tính, thực hiện nguyên tắc tối thiểu và tối đa với alpha-beta clipping. So sánh với tùy chọn không bị cắt ngắn alpha-beta.

6.5.4. Duyệt alpha-beta đến một độ sâu nhất định

Chúng ta sẽ tập trung lại vào ví dụ về cờ vua. Để thực hiện duyệt thông tin alpha-beta nhằm xác định điểm cao nhất, chúng ta sẽ phải duyệt từ một cái cây khổng lồ, nơi ngay cả những tối ưu hóa như nhát cắt alpha-beta cũng không giúp ích gì đáng kể cho chúng ta. Thường trong những trường hợp như vậy, việc duyệt đến một độ sâu nhất định được áp dụng. Khi đạt đến điểm đó, quá trình duyệt bị dừng lại và việc đánh giá đỉnh điểm mà nó bị gián đoạn được tính theo một số tiêu chí khác. Đối với cờ vua, tiêu chí này có thể là, quân nào còn lại trên bàn cờ, quyền tự do di chuyển của họ, ai là người sở hữu tâm bàn cờ, v.v. Hàm tính toán ước tính cấu hình sẽ không yêu cầu nhiều thời gian tính toán, vì nó sẽ được thực hiện thường xuyên. Một cách có thể để ước tính là bằng cách so sánh các giá trị của các số liệu [Brassard, Bratley - 1996]:

Quân Hậu - 10; Quân xe 5; Tượng, Ngựa - 3,25; Quân Tốt - 1;

Đánh giá một cấu hình (tích cực hoặc tiêu cực) có thể được định nghĩa là tổng điểm của các quân trắng trừ đi tổng điểm của các quân đen. Đối với các vị trí đầu cuối, chúng ta giả định rằng khi quân đen ở vị trí mờ, điểm cấu hình là $+\infty$ và khi người da trắng mờ, điểm là $-\infty$. Khi trò chơi kết thúc với tỷ số hòa - giá trị là 0.

Hàm ước lượng đỉnh tương tự như trong 6.5.2 với sự khác biệt duy nhất mà nó gây ra khi tính toán giá trị cấu hình khi đạt đến độ sâu nhất định:

Hàm tính giá trị theo độ sâu

```

value eval(i) {
    return <Tính theo tiêu chuẩn cho cấu hình>;
}

value minimax(i, depth) {
    if (<i là đỉnh lá>) return <đánh giá của đỉnh lá ($+\infty,-\infty,
    0$);
    if (depth > maxDepth) return eval(i); /* tính toán ràng buộc */
        <lấy (i1 , i2 , ... , in ) là các thừa kế của i>;
    if (<i là đỉnh lấy min>)
        return min(minimax(i1 ,depth+1), .., minimax(in ,depth+1));
    if (<i là đỉnh lấy max>)
        return max(minimax(i1 ,depth+1), .., minimax(in ,depth+1));
}

```

Độ sâu của maxDepth thu thập thông tin càng lớn thì ước tính càng chính xác. Tuy nhiên, chúng ta sẽ không bao giờ chắc chắn rằng đánh giá là chính xác tuyệt đối. Ví dụ: luôn có thể có một quá trình phát triển trò chơi hy sinh một con số trong cấu hình độ sâu maxDepth (điều này sẽ làm giảm xếp hạng eval (i)) để có được vị trí tốt hơn: Trong một số bước tiếp theo, thậm chí maxDepth + 1), mà sẽ không được xem xét, có thể giành chiến thắng một con số mạnh hơn hoặc thậm chí kiểm tra đối thủ.

Cùng với việc thu thập thông tin alpha-beta, bạn có thể áp dụng phương pháp nhát cắt alpha-beta. Vào năm 1997, tốc độ của máy tính DeepBlue, đã đánh bại nhà vô địch thế giới, cho phép 200.000.000 lượt kiểm tra cấu hình cờ vua mỗi giây. Với việc bổ sung thêm alpha-beta clipping, độ sâu của cây mà máy tính này có thể khám phá là khá cao. Ngoài ra, hàm eval() cực kỳ chính xác (một số người chơi cờ chuyên nghiệp đã thực hiện điều này). Có thể nói, phần cứng máy tính hiện đã qua thời điểm phát triển quan trọng, lúc này người ta không còn có thể cạnh tranh trong trò chơi cờ vua với một chương trình cờ vua được thiết kế tốt.

Bài tập

- 6.43. Hãy đề xuất phương án đánh giá chất lượng cấu hình cho các trò chơi "X" và "O".

► **6.44.** Để xuất ra một biến thể của trò chơi "X" và "O" bằng cách sử dụng nhát cắt alpha-beta đến một độ sâu nhất định.

6.6. Câu hỏi và bài tập

Trong phần cuối của chương, chúng ta sẽ đưa ra danh sách các bài toán NP-đầy đủ nổi tiếng nhất, kèm theo lời giải thích ngắn gọn về một số trong số đó. Các bài toán *NP*-đầy đủ thường xuất hiện trong thực tế và việc nhận ra chúng như vậy có thể giúp chúng ta tiết kiệm rất nhiều công sức. Thực tế là một bài toán *NP*-đầy đủ sẽ cho phép chúng ta không lãng phí thời gian để tìm kiếm một thuật toán nhanh, mà tập trung vào một trong những cách tiếp cận tiêu chuẩn sau để giải quyết:

- *Sử dụng thuật toán heuristic* (xem Chương 9). Nếu vấn đề không thể được giải quyết nhanh chóng trong mọi trường hợp, thì vẫn có thể có một phương pháp nhanh chóng chỉ có thể giải quyết trong một số trường hợp.
- *Giải quyết vấn đề gần đúng* (xem Chương 9). Đối với một số bài toán *NP*-đầy đủ, các thuật toán xấp xỉ tồn tại. Họ sẽ không giải quyết vấn đề một cách chính xác, nhưng họ sẽ luôn tìm ra một giải pháp có thể được chứng minh là đủ gần.
- Nhiều bài toán thực tế, vốn có tính chất *NP*-đầy đủ, có thể được chia thành một số trường hợp *đặc biệt chính*. Trong một số trường hợp, có thể có một thuật toán phức tạp đa thức sử dụng, ví dụ, bộ nhớ bổ sung (tối ưu hóa động - Chương 8).
- *Sử dụng vét kiệt hoàn toàn*. Ở đây, cách nhận thức cũng rất quan trọng. Đối với một số trường hợp trong bài toán có thể chắc chắn rằng chúng sẽ không dẫn đến một giải pháp cho phép loại trừ chúng khỏi quá trình nghiên cứu và do đó làm giảm độ phức tạp về thời gian. Những điều này đã được thảo luận trong chương này.

Nhiều bài toán chúng ta sẽ xem xét được xuất bản lần đầu tiên vào năm 1979 trong "bách khoa toàn thư" về các bài toán *NP*-đầy đủ - [Garey, Johnson-1979]. Để đầy đủ, danh sách cũng chứa các bài toán đã được xem xét (một số trong số chúng đã được giải quyết). Mỗi bài toán dưới đây được thảo luận trong sơ đồ sau:

- *Tên của bài toán* (và tên viết tắt, nếu biết). Đây là tên mà bài

toán có thể được tìm thấy thường xuyên nhất trong tài liệu.

- *Điều kiện của bài toán.* Nó bao gồm hai phần: dữ liệu đầu vào (thường chúng là các cấu trúc cụ thể - đồ thị, tập hợp, biểu thức logic, v.v.) và những gì bạn đang tìm kiếm. Câu hỏi thứ hai được xây dựng như một câu hỏi mà đối với các đầu vào cụ thể, thuật toán phải trả lời có hoặc không.
- *Nhận xét về bài toán.* Phần này (không phải lúc nào cũng có) bao gồm một bình luận ngắn về bài toán, có thể bao gồm các hướng dẫn (hoặc các cách tiếp cận ít được biết đến hơn) để giải quyết bài toán, các biến thể, ứng dụng của nó, v.v. Ở đây có thể có các tài liệu tham khảo khác liên quan đến bài toán. Như chúng ta đã nói, nhiều bài toán trong danh sách này có thể được tìm thấy được phân loại trong [Garey, Johnson-1979], vì vậy chúng ta sẽ không đề cập đến cuốn sách này trong từng bài toán cụ thể. Sau đây là một ví dụ:

▷ 6.45. Đồ thị 3 màu (3-COL)

- Một đồ thị được đưa ra.
- Kiểm tra xem có thể "tô màu" mỗi đỉnh của đồ thị (so sánh nó) với một trong ba màu: lục, lam, đỏ, sao cho không có hai đỉnh liền kề nào được tô cùng màu.

Vì vậy, đặt ra, bài toán liên quan đến việc tìm ra lời giải cho một đồ thị tùy ý. Điều kiện chỉ muốn kiểm tra xem có thể tô màu cho cột 3 màu hay không. Đối với nhiều bài toán trong danh sách này, điều kiện có thể được tóm tắt để tìm ra giải pháp tối ưu - đối với ví dụ cụ thể, có thể tìm cách tô màu với số lượng màu tối thiểu, v.v.

Hơn nữa, các bài toán theo sau với độ khó tăng dần. Các bài toán từ đầu danh sách được đặt ở dạng đơn giản hơn, chúng liên quan đến một bộ máy toán học đơn giản hơn, được thảo luận rộng rãi trong các tài liệu. Tình huống hoàn toàn ngược lại là với các bài toán ở cuối danh sách - điều kiện phức tạp hơn và ít được biết đến hơn.

▷ 6.46. Đường Hamilton trong đồ thị

- Một đồ thị không định hướng được đưa ra.
- Kiểm tra xem có một đường đi đơn giản nào chứa tất cả các đỉnh của đồ thị hay không.
- Bài toán được xem xét trong: ??, 6.2, ?? và những người khác.

▷ 6.47. Chu trình tối đa

- Một đồ thị không định hướng với n đỉnh và số tự nhiên $k, 1 \leq k \leq n$ là cho trước.
- Kiểm tra xem có một vòng lặp đơn giản trong đồ thị chứa ít nhất k đỉnh hay không.
- Đây là bản tóm tắt của bài toán tìm đường đi Hamilton. Trong thực tế, bài toán kiểm tra xem một đồ thị có chu trình Hamilton hay không là một trường hợp đặc biệt (nếu chúng ta đặt $k = n$).

▷ 6.48. Sự thỏa mãn của một hàm Boolean

- Thứ m loại bỏ C_1, C_2, \dots, C_m của các biến Boolean X_1, X_2, \dots, X_n và các số phủ định của chúng được đưa ra.
- Để kiểm tra xem có sự gán giá trị "true" hoặc "false" cho các biến X_1, X_2, \dots, X_n , mà giá trị của tất cả các liên từ C_1, C_2, \dots, C_m là "true".
- Chúng ta đã giải quyết vấn đề trong 6.3.1.

▷ 6.49. Con đường dài nhất

- Một đồ thị không định hướng với n đỉnh và số tự nhiên $k, 1 \leq k \leq n$ là cho trước. Hai đỉnh s, t của đồ thị là cố định.
- Kiểm tra xem có một đường đi đơn giản từ s đến t trong đó ít nhất k cạnh tham gia hay không.
- Chúng ta đã giải quyết một biến thể của bài toán trong 6.3.3.

▷ 6.50. Một bài toán cho khách du lịch

- Một đồ thị có trọng số không định hướng với n đỉnh và số tự nhiên $k, k \geq 1$ đã cho.
- Kiểm tra xem có một chu trình đơn giản bao gồm tất cả các đỉnh của đồ thị, với độ dài (là tổng trọng lượng của các sườn) nhiều nhất là k .
- Đây là một dạng biến thể của bài toán nổi tiếng về tìm chu trình Hamilton nhỏ nhất trong đồ thị, mà chúng ta đã xem xét nhiều lần.

▷ 6.51. Bài toán ba lô

- Một tập hợp hữu hạn A được cho trước và một số tự nhiên $s(a)$ được ánh xạ tới mỗi phần tử $a \in A$. Số tự nhiên k đã cho.

- Kiểm tra xem có tồn tại một tập con $B \subseteq A$ sao cho tổng các giá trị $s(x)$ của $x \in B$ là chính xác k .
- Đây là bài toán 1 của 6.4.1. Chúng ta quyết định biến thể tối ưu hóa của nó theo phương pháp phân nhánh và ranh giới. Chúng ta sẽ xem xét nó một lần nữa trong ??, Nơi chúng ta sẽ giải quyết nó với tối ưu hóa động.

▷ 6.52. Bài toán của Alan và Bob

- Tập hợp hữu hạn A . Cho trước, một số tự nhiên $s(a)$ được ánh xạ tới mỗi phần tử a .
- Kiểm tra xem A có thể chia thành hai tập A_1 và A_2 hay không mà $\sum_{x \in A_1} s(x) = \sum_{y \in A_2} s(y)$.
- Tên của bài toán "Alan và Bob" xuất phát từ bài toán thực tế là chia một số phần quà (mỗi phần quà có giá trị nhất định) giữa hai anh em (Alan và Bob) sao cho tổng giá trị của các phần quà của một là bằng tổng từ các giá trị của quà tặng của người kia. bài toán được xem xét trong ??.

▷ 6.53. Tích từ một tập hợp con.

- Một tập hợp hữu hạn A được cho trước và một số tự nhiên $s(a)$ được ánh xạ tới mỗi phần tử $a \in A$. Số tự nhiên k đã cho.
- Kiểm tra xem có tồn tại tập con $B \subseteq A$ sao cho tích của các phần tử $s(x)$, $x \in B$ chính xác bằng k .
- Đây là một biến thể của bài toán ba lô.

▷ 6.54. Đóng gói hộp

- Tập hợp A hữu hạn có n phần tử đã cho. Một số tự nhiên $s(a)$ được ánh xạ tới mỗi phần tử $a \in A$. Các số tự nhiên b và k đã cho, $1 \leq k \leq n$.
- Kiểm tra xem A có thể chia thành k tập A_1, A_2, \dots, A_k sao cho với mỗi A_i ($1 \leq i \leq k$), tổng các giá trị của các phần tử của nó không vượt quá b .
- Bài toán là một bản tóm tắt của bài toán cho ba lô - k ba lô được đưa ra, mỗi ba lô có thể chứa b kg và phải phân phối các vật dụng trong đó. Một biến thể tối ưu hóa của vấn đề cũng có thể xảy ra - để tìm k tối thiểu mà các đối tượng có thể được phân phối theo cách được mô tả.

▷ 6.55. Phủ hình vuông

- N màu (ký hiệu là số nguyên $1, 2, \dots, n$) và danh sách các ô vuông tô màu có một cạnh được đưa ra. Mỗi hình vuông được cho bởi bốn (a, b, c, d) cho biết mặt trên tô màu a , mặt dưới tô màu b , mặt trái tô màu c và mặt bên phải tô màu d . Số tự nhiên k đã cho, $1 \leq k \leq n$.
- Kiểm tra xem có một ô vuông có kích thước $k \times k$ được tô bằng cách sử dụng các ô vuông đã cho để các mặt tiếp xúc của chúng có cùng màu hay không.
- [Kelevedzhiev-5/1998]

▷ 6.56. Trò chơi ô chữ

- Tập hợp n ký hiệu $S = \{s_1, s_2, \dots, s_n\}$ và tập $W = \{w_1, w_2, \dots, w_{2n}\}$ gồm $2n$ từ được đưa ra sao cho mỗi w_i là một dãy gồm đúng n ký tự từ S .
- Để kiểm tra xem có thể thu được ô chữ có kích thước $n \times n$ từ $2n$ từ đã cho hay không, tức là nếu C là ma trận có kích thước $n \times n$, thì ký hiệu của S có thể được viết trong mỗi ô sao cho mỗi dòng và một cột của ma trận để biểu diễn một từ từ W .

▷ 6.57. Tập độc lập trong đồ thị

- Một đồ thị không định hướng $G(V, E)$ với n đỉnh và số tự nhiên k , $1 \leq k \leq n$ là cho trước.
- Kiểm tra xem có tồn tại một tập con $U \subseteq V$ chứa ít nhất k đỉnh trong đó không tồn tại hai đỉnh $i, j \in U$ sao cho $(i, j) \in E$.
- Một biến thể của bài toán được xem xét trong 5.7.2.

▷ 6.58. Đồ thị không có hình tam giác

- Một đồ thị không định hướng $G(V, E)$ được cho.
- Để kiểm tra xem có thể chia tập E thành hai tập con E_1 và E_2 sao cho không trong chúng có ba bộ lạc có dạng $(i, j), (j, k), (k, i)$ (nghĩa là trong hai đồ thị $G_1(V, E_1), G_2(V, E_2)$ không có chu trình nào có độ dài 3).

▷ 6.59. Bộ thống trị

- Một đồ thị không định hướng $G(V, E)$ với n đỉnh và số tự nhiên k , $1 \leq k \leq n$ là cho trước.

- Kiểm tra rằng G có chứa tập con U , $U \subseteq V$ với nhiều nhất k đỉnh và sao cho mỗi đỉnh $i \in V$, $i \in U$ tồn tại một đỉnh $j \in U$ sao cho $(i, j) \in E$.

▷ 6.60. Số màu cạnh

- Đồ thị không định hướng $G(V, E)$ với m cạnh và số tự nhiên k , $1 \leq k \leq m$ là cho trước.
- Kiểm tra xem có thể so sánh một số tự nhiên duy nhất từ 1 đến k trên mỗi sườn hay không (tô màu cho sườn với màu từ 1 đến k) để không có hai sườn ngẫu nhiên nào có màu giống nhau.
- Định lý Vizing chỉ ra rằng trường hợp phức tạp duy nhất trong bài toán này là khi k bằng hoành độ của đỉnh của đồ thị. Tính hoàn chỉnh NP của bài toán này đã được chứng minh gần đây. Có rất nhiều thuật toán heuristic và song song để giải nó [Gibbons, Rytter-1987].

▷ 6.61. Chuỗi con chung ngắn nhất

- Cho tập $S = \{s_1, s_2, \dots, s_n\}$ gồm các xâu nhị phân (dãy 0 và 1) và một số tự nhiên k .
- Kiểm tra xem có tồn tại một chuỗi s có độ dài tối đa k sao cho mỗi chuỗi $p \in S$ là một chuỗi con của s hay không.
- Tóm tắt của bài toán là khi nhiều nhân vật khác nhau có thể tham gia vào các chuỗi, nhưng ngay cả trong trường hợp đơn giản này, bài toán vẫn hoàn thành NP.

▷ 6.62. Bài toán của bưu tá

- Cho đồ thị $G(V, E)$ với m cạnh, tập con E_1 , $E_1 \subseteq E$ và số tự nhiên k , $1 \leq k \leq m$.
- Kiểm tra xem có chu trình nào trong cột (không nhất thiết phải đơn giản) chứa mỗi cạnh $e \in E_1$ ít nhất một lần và có tổng số cạnh nhỏ hơn hoặc bằng k .

▷ 6.63. Bộ phận "cấp trên"

- Hai dãy số tăng nghiêm ngặt $A = < a_1, a_2, \dots, a_n >$ và $B = < b_1, b_2, \dots, b_m >$ của các số nguyên dương được cho.
- Với $\text{Div}(x, Y)$ (Y là một dãy số tự nhiên), chúng ta sẽ ký hiệu số phần tử $y \in Y$, sao cho x là ước chính xác của y . Kiểm tra xem có tồn tại số tự nhiên c mà $\text{Div}(c, A) > \text{Div}(c, B)$ hay không.

- Đây và ba bài toán tiếp theo là một loại đặc biệt của bài toán NP -đầy đủ - từ lập trình số nguyên. Trong bài toán cụ thể, chúng ta có thể tìm kiếm c một cách tuần tự, bắt đầu từ 1. Tuy nhiên, thuật toán này không hiệu quả và bài toán là NP -đầy đủ. Điều này đáng chú ý nếu chúng ta so sánh kích thước của dữ liệu đầu vào (số bit để biểu diễn số trong hệ thống số nhị phân) với độ phức tạp của thuật toán trong trường hợp xấu nhất - chúng ta có một hàm số mũ.

▷ 6.64. Phương trình Diophantine bậc hai

- Các số tự nhiên a, b, c đã cho.
- Kiểm tra xem có tồn tại các số nguyên x và y sao cho $(a \cdot x^2) + (b \cdot y) = c$.
- Nhận xét tương tự áp dụng ở đây như trong Bài toán 19 - tìm kiếm tuần tự cho các số x và y có độ phức tạp theo cấp số nhân bằng số bit cần thiết để biểu diễn các số a, b và c .

CHƯƠNG 7

THUẬT TOÁN CHIA ĐỂ TRỊ

7.1. Giới thiệu	560
7.2. Tìm phần tử lớn nhất thứ k	561
7.3. Phần tử trội	572
7.4. Hợp nhất các mảng đã sắp xếp	590
7.5. Sắp xếp theo hợp nhất	597
7.6. Nâng nhanh lũy thừa	607
7.7. Thuật toán Strassen để nhân nhanh các ma trận	610
7.8. Nhân nhanh các số dài	616
7.9. Bài toán tháp Hà Nội	620
7.10. Tổ chức giải vô địch bóng đá	624
7.11. Sự dịch chuyển tuần hoàn của các phần tử mảng	631
7.12. Câu hỏi và bài tập	636

7.1. Giới thiệu

Nguồn gốc của ý tưởng chia bài toán phức tạp thành nhiều bài toán đơn giản hơn, dễ tấn công riêng lẻ hơn và giải pháp của nó cho phép dễ dàng xây dựng giải pháp cho vấn đề ban đầu, đã trở lại rất xa xưa. Nó đạt đến đỉnh cao trong thời kỳ Đế chế La Mã, nơi đã hình thành và thúc đẩy "Chia rẽ và Chinh phục" như một nguyên tắc cơ bản trong chính sách đối ngoại của mình đối với các bộ tộc láng giềng của đế chế. Trên thực tế, người La Mã hoàn toàn không phải là người khám phá ra nguyên tắc, và nó là trọng tâm của sự bành trướng của bất kỳ đế chế vĩ đại nào trước và sau thời La Mã.

Làm thế nào là mọi thứ trong lập trình? Việc áp dụng phương pháp diễn ra trong ba bước. Đầu tiên, bài toán ban đầu được chia thành nhiều bài toán con, thường là hai (chia). Nó sau để giải quyết từng người trong số họ riêng biệt. Trong bước thứ ba, dựa trên các giải pháp của các bài toán con, một giải pháp của bài toán ban đầu (chính) được xây dựng.

Tất nhiên, nguyên tắc này không phải là phổ biến và việc áp

dụng nó không phải là khả thi và thích hợp trong mọi công việc. Các lý do tại sao không thể áp dụng phương pháp này có thể khác nhau. Ví dụ: bài toán có thể không cho phép chia nhỏ các bài toán con thích hợp có thể được giải quyết riêng lẻ. Có thể giải pháp của các bài toán phụ riêng lẻ không tạo thuận lợi đáng kể cho giải pháp của bài toán ban đầu hoặc thậm chí không liên quan gì đến nó. Mặc dù ý tưởng của phương pháp này rất đơn giản, nhưng việc áp dụng nó, cũng như xác định khi nào nó có thể được áp dụng, là một môn nghệ thuật mà bạn dễ dàng nắm vững nhất dựa trên các ví dụ cụ thể.

7.2. Tìm phần tử lớn nhất thứ k

Một trong những thuật toán đầu tiên được giới thiệu cho lập trình viên mới là tìm phần tử tối đa của một mảng. Đây là một thuật toán khá đơn giản yêu cầu so sánh $n - 1$:

Chương trình 7.1. Tìm phần tử lớn nhất trong (701maximum.c)

```
int findMax(int m[], unsigned n) /* Tìm phần tử lớn nhất */
{
    unsigned i;
    int max;
    for (max = m[0], i = 1; i < n; i++)
        if (m[i] > max) /**
            max = m[i];
    return max;
}
```

Chúng ta có thể cải thiện kết quả này không? Không khó để thuyết phục rằng thuật toán được đề xuất về mặt lý thuyết thực hiện số phép so sánh phần tử tối thiểu. Điều sau rất dễ nhận thấy nếu chúng ta nghĩ về các yếu tố là những người tham gia vào một giải đấu loại trực tiếp. Thật vậy, để xác định người thắng cuộc, mỗi người tham gia, trừ chính xác một (người thắng cuộc), phải thua ít nhất một lần để bị loại.

Từ đó có thể dễ dàng thấy rằng số cuộc họp cần thiết là $n - 1$. Ngoài số lượng phép so sánh, một tính năng quan trọng của bất kỳ thuật toán nào như vậy là số lần gán giá trị. Từ mã của đoạn chương trình được đề xuất, có thể thấy ngay rằng số lượng bài tập không

vượt quá số lượng phép so sánh. Tuy nhiên, không quá rõ ràng con số trung bình của các vụ biến thủ là bao nhiêu. Chúng ta để người đọc chứng tỏ rằng nó có bậc là $\Theta(\log_2 n)$.

Hãy tưởng tượng một chương trình để hình dung một tập hợp các dấu chấm trên màn hình máy tính. Vì số lượng và vị trí của các chấm có thể rất khác nhau, chương trình sẽ cần chia tỷ lệ tọa độ của chúng theo một cách nào đó để chúng trông đẹp trên màn hình. Màn hình có dạng hình chữ nhật nên sẽ rất tiện lợi khi “đóng” các điểm lại trong một hình chữ nhật phù hợp, từ đó tính ra tỉ lệ tương ứng. Rõ ràng là hình chữ nhật này được xác định bởi các tọa độ x và y lớn nhất và nhỏ nhất. Ở đây nảy sinh đồng thời bài toán tìm phần tử tối thiểu và tối đa.

Bài toán mới cũng có một phương pháp giải tuyến tính. Cần so sánh $2n - 1$ để tìm ra phần tử lớn nhất và sau đó/trước và nhỏ nhất một cách độc lập theo thuật toán trên (Khi tìm kiếm phần tử nhỏ nhất, chỉ cần xoay dấu so sánh theo thứ tự được đánh dấu bằng / * */.). Tuy nhiên, giải pháp này là không tối ưu, vì trong quá trình tìm kiếm giá trị tối thiểu, các tỷ lệ đã biết thu được trong tìm kiếm giá trị tối đa không được tính đến.

Dưới đây, chúng ta sẽ trình bày một thuật toán khác, trong trường hợp xấu nhất, chúng ta sẽ không cần nhiều hơn $\lceil 3n/2 \rceil$ phép so sánh. Chúng ta sẽ xem xét các phần tử theo từng cặp: đầu tiên chúng ta sẽ so sánh hai phần tử với nhau, sau đó phần tử lớn hơn với mức tối đa hiện tại và phần tử nhỏ hơn với mức tối thiểu hiện tại, thực hiện ba so sánh cho mỗi cặp phần tử.

Tìm đồng thời max và min trong 701maximum.c

```
void findMinMax(int *min, int *max, const int m[], const
                unsigned n)
/*Tìm đồng thời max và min*/
{ unsigned i, n2;
  for (*min = *max = m[n2 = n/2], i = 0; i < n2; i++)
    if (m[i] > m[n-i-1]) {
      if (m[i] > *max) *max = m[i];
      if (m[n-i-1] < *min) *min = m[n-i-1];
    }
    else {
      if (m[n-i-1] > *max) *max = m[n-i-1];
    }
}
```

```

    } if (m[i] < *min) *min = m[i];
}

```

Mọi thứ như thế nào nếu chúng ta đang tìm kiếm phần tử lớn thứ hai? Rõ ràng, chúng ta có thể tận dụng cùng một chiến lược, sửa đổi chức năng một cách thích hợp để duy trì không chỉ một, mà cả hai giá trị tốt nhất. Ý tưởng rất đơn giản: chúng ta sẽ so sánh phần tử tiếp theo với phần tử lớn thứ hai hiện tại và chỉ khi phần tử lớn hơn được tìm thấy, chúng ta sẽ thực hiện kiểm tra bổ sung xem phần tử đó không lớn hơn phần tử lớn nhất hiện tại hay chưa:

Tìm phần tử max thứ hai trong 701maximum.c

```

void swap(int *el1, int *el2) /* Hoán đổi giá trị */
{ int tmp = *el1; *el1 = *el2; *el2 = tmp; }

int findSecondMax(int m[], unsigned n)
{ int x,y;
  unsigned i;
  x = m[0]; y = m[1];
  if (y > x) swap(&x,&y);
  for (i = 2; i < n; i++)
    if (m[i] > y)
      if ((y = m[i]) > x) swap(&x,&y);
  return y;
}

```

Hãy thử ước tính độ phức tạp của thuật toán. Không khó để thấy rằng trường hợp xấu nhất là mảng nghịch đảo, trong đó hai phép so sánh được thực hiện trên mỗi $(n - 2)$ lần lặp của chu kỳ. Thêm phép so sánh trước chu kỳ, với tổng số phép so sánh cần thiết, chúng ta nhận được $2n - 3$.

Bây giờ chúng ta hãy xem xét vấn đề tổng quát hơn là tìm *phần tử nhỏ nhất thứ k* của một tập hợp n phần tử đã cho. Bài toán được xem xét tương đương với việc tìm phần tử $(n - k + 1)$ lớn nhất. Hơn nữa, nếu chúng ta có một thuật toán giải quyết vấn đề ban đầu, đảo ngược các so sánh $<$ (hoặc \leq) và $>$ (hoặc \geq), với những sửa đổi tối thiểu, chúng ta có thể tìm thấy phần tử lớn nhất thứ k.

Một giải pháp nhỏ cho vấn đề là sắp xếp hoàn chỉnh tập hợp,

kết quả là phần tử thứ k sẽ chiếm vị trí thứ k . Nhược điểm chính của phương pháp này là, bất kể giá trị của k là bao nhiêu, nó yêu cầu một số phép so sánh bậc $n \cdot \log_2 n$. Tất nhiên, chúng ta có thể giảm con số này xuống $\Theta(n + k \cdot \log_2 n)$, chỉ sắp xếp k phần tử đầu tiên bằng cách sắp xếp theo hình chóp (việc xây dựng kim tự tháp cần thời gian theo thứ tự của $\Theta(n)$ và sắp xếp theo hình chóp có độ phức tạp $\Theta(n \cdot \log_2 n)$ cả trong trường hợp trung bình và trong trường hợp xấu nhất, xem ??). Phát triển thêm ý tưởng, chúng ta thu được độ phức tạp $\Theta(\min + (n + k \cdot \log_2 n, n + (n - k + 1) \cdot \log_2 n))$. Thật vậy, nếu $k > [n/2]$, chúng ta có thể đảo ngược hướng sắp xếp theo thứ tự giảm dần. Tuy nhiên, nếu k đủ gần với $n/2$, thuật toán kết quả rõ ràng sẽ không hiệu quả, vì nó sẽ dẫn đến việc sắp xếp hoàn toàn k hoặc $(n - k + 1)$ phần tử của mảng, điều này sẽ dẫn đến một lượng lớn thông tin không cần thiết.

Chương trình 7.2. Tìm phần tử thứ k (702heap.c)

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
int m[MAX];
const unsigned n = 100; /* Số phần tử trong mảng */
const unsigned k = 10; /* Số thứ tự của phần tử được tìm kiếm */

void init(int m[], unsigned n) /* Khởi tạo với số ngẫu nhiên */
{
    unsigned i;
    for (i = 0; i < n; i++)
        m[i] = rand() % (2*n + 1);
}

void siftMin(int l, int r) /* Sàng lọc phần tử từ đỉnh tự tháp*/
{
    int i, j;
    int x;
    i = l;
    j = i + i + 1;
    x = m[i];
    while (j <= r) {
        if (j < r)
            if (m[j] > m[j+1])
                j++;
        ...
    }
}
```

```

if (x <= m[j])
    break;
m[i] = m[j];
i = j;
j = j*2 + 1;
}
m[i] = x;
}

void siftMax(int l,int r) /* Sàng lọc phần tử từ đỉnh tự tháp */
{
    int i,j;
    int x;
    i = 1; j = i + i + 1; x = m[i];
    while (j <= r) {
        if (j < r)
            if (m[j] < m[j+1]) j++;
        if (x >= m[j])
            break;
        m[i] = m[j];
        i = j;
        j = j*2 + 1;
    }
    m[i] = x;
}

void heapFindK(unsigned k) /* Tìm phần tử thứ k từ tự tháp*/
{
    int l,r;
    char useMax;
    if (useMax = (k > n/2))
        k = n - k - 1;
    l = n/2; r = n - 1;
    /* Xây dựng tự tháp */
    while (l-- > 0)
        if (useMax) siftMax(l,r); else siftMin(l,r);
    /* (k-1)-bội lần bỏ phần tử nhỏ nhất */
    for (r = (int)n-1; r >= (int)(n-k); r--) {
        m[0] = m[r];
        if (useMax) siftMax(0,r); else siftMin(0,r);
    }
}

```

```

void print(int m[], unsigned n) /* biểu diễn mảng lên màn hình */
{
    unsigned i;
    for (i = 0; i < n; i++)
        printf("%8d", m[i]);
}

int main() {
    init(m,n);
    printf("Mảng trước khi tìm:"); print(m,n);
    printf("\nTìm phần tử thứ k: k=%u", k);
    heapFindK(k);
    printf("\nPhần tử thứ k là: %d", m[0]);
    return 0;
}

```

Trong một số điều kiện ràng buộc mạnh hơn, chúng ta có thể dễ dàng thu được các thuật toán tuyến tính thậm chí. Ví dụ, bằng cách sử dụng ý tưởng của thuật toán đếm (xem 3.2.2.), Chúng ta có thể tìm số lần xuất hiện c_j cho mỗi khóa j trong số các giá trị cho phép của các khóa. Giả sử rằng các khóa là các số nguyên trong khoảng $[l, r]$. Sau đó ta có thể tính tổng liên tiếp $c_l, c_{l+1}, c_{l+2}, \dots$ cho đến khi đạt được chỉ số j mà tổng của lần đầu tiên trở nên lớn hơn hoặc bằng k . Phần tử ở giữa bắt buộc sẽ có giá trị là j . Tuy nhiên, một lượng lớn thông tin dư thừa lại tích lũy và độ tuyến tính của thuật toán phụ thuộc rất nhiều vào sự phân bố của các giá trị khóa.

Mặc dù thoạt nhìn điều này không rõ ràng, nhưng ý tưởng sắp xếp nhanh (xem 3.1.6.) Để chia một mảng phân vùng cung cấp một thuật toán hiệu quả để giải quyết vấn đề. Đây là một thuật toán chia để trị được đề xuất bởi Hoar, tác giả của sắp xếp nhanh (cũng dựa trên phép chia để trị).

Mảng được chia thành hai phần với biên trái $l = 0$, biên phải $r = n - 1$ và $x = m[k]$ như một phần tử ngăn cách (chúng ta giả sử rằng mảng được lập chỉ mục từ 0 đến $n - 1$). Sau khi chia theo thuật toán sắp xếp nhanh cho các chỉ số i và j , các bất đẳng thức được thỏa mãn:

$$m[h] \leq x, \text{ cho } h < i$$

$m[h] \geq x$, cho $h > j$

$i > j$

Ba trường hợp có thể xảy ra:

- $i < k$. Sau đó x chia mảng cho ít hơn yêu cầu. Việc tách nên được tiếp tục bằng cách đặt $l = i$. (xem Hình 7.2)



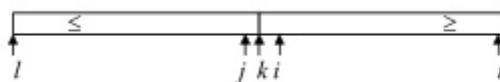
Hình 7.1. Mảng trong trường hợp $i < k$

- $j > k$. Sau đó x chia mảng theo tỷ lệ lớn hơn giá trị được tìm. Việc tách nên được tiếp tục bằng cách đặt $r = j$. (xem Hình ??)



Hình 7.2. Mảng trong trường hợp $j > k$

- $j < k < i$. Sau đó x chia mảng thành các tỷ lệ bắt buộc và do đó là phần tử bắt buộc. Thật vậy, tất cả các phần tử bên trái của nó đều nhỏ hơn hoặc bằng nó, và những phần tử ở bên phải lớn hơn hoặc bằng nó. Quá trình phân tách được kết thúc. (xem Hình 7.3)



Hình 7.3. Mảng trong trường hợp $j < k < i$

Chúng ta để người đọc thấy rằng các trường hợp khác về vị trí tương hỗ của i, j và k là không thể. Khi đạt đến trường hợp 3, phần tử thứ k sẽ ở vị trí thứ k trong mảng.

Chương trình 7.3. Tìm phần tử thứ k (703midelem.c)

```

void find(int m[],unsigned n,unsigned k) /*Tìm phần tử thứ k*/
{
    int i,j,l,r;
    int x;
    l = 0; r = n - 1;
    while (l < r) {
        x = m[k]; i = l; j = r;
        for(;;) {
            while (x > m[i]) i++;
            while (x < m[j]) j--;
            if (i > j)
                break;
            swap(m + i, m + j);
            i++;
            j--;
        }
        if (j < (int)k)
            l = i;
        if ((int)k < i)
            r = j;
    }
}

```

Thuật toán được mô tả yêu cầu một số phép so sánh theo bậc của $2n$, trong trường hợp ở mỗi bước, phân hoạch chứa phần tử thứ k bị giảm đi một nửa. Chất lượng này xác định nó hiệu quả hơn những chất lượng được đề xuất ở trên, dựa trên cách sắp xếp hình chóp, yêu cầu một số so sánh về thứ tự của $\Theta(\min(n + k \log_2 n, n + (n - k + 1)) \cdot \log_2 n))$, cả trong trường hợp xấu nhất và trong trường hợp tốt nhất. Thật không may, trong trường hợp xấu nhất, khi tại mỗi bước diện tích chỉ giảm đi 1, độ phức tạp của nó sẽ có bậc là $\Theta(n^2)$. Do đó, kế thừa nhược điểm chính của sắp xếp Hoor nhanh, mặc dù nói chung là rất tốt, thuật toán được mô tả ra cực kỳ kém hiệu quả trong trường hợp xấu nhất.

Chúng ta sẽ đề xuất một thuật toán đệ quy khác, lần này dựa trên phân loại Hoor nhanh với các đặc điểm tương tự. Chúng ta sẽ chia đôi với một số phần tử x , bắt kể phần tử nào ($m[r]$) được chọn trong chương trình bên dưới) và để nó rơi xuống vị trí mid từ đầu mảng hoặc đến vị trí p từ đầu trong tổng số cổ phần đang được xem

xét. Nếu $k \leq p$, thì chúng ta nên tìm phần tử thứ k trong phân hoạch bên trái (l, mid) . Nếu không, chúng ta nên tìm phần tử $(k - p)$ trong phân vùng bên phải $(mid + 1, r)$. Quá trình tiếp tục đệ quy cho đến khi đạt được một phân vùng chứa một phần tử (cụ thể là các tìm kiếm):

Chương trình 7.4. Tìm theo Hoor (704midel2.c)

```

unsigned partition(unsigned l,unsigned r) /* Phân chia theo Lomuto
/*
{ int i;
  unsigned j;
  int x;
  i = l - 1; x = m[r];
  for (j = 1; j <= r; j++)
    if (m[j] <= x) {
      i++;
      swap(m+i,m+j);
    }
  if (i == (int)r) /* Tất cả <= x. Thu hẹp khu vực với l. */
    i--;
  return i;
}

unsigned find(int l, int r, unsigned k) /* T theo Hoor */
{
  unsigned mid, p;
  if (l == r) return l;
  mid = partition(l,r);
  p = mid - 1 + 1;
  return k < p ? find(l,mid,k) : find(mid+1,r,k-p);
}

```

Thuật toán được mô tả có cùng hành vi với thuật toán trước: rất tốt trong trường hợp chung và cực kỳ kém hiệu quả trong trường hợp xấu nhất. Hành vi xấu của cả hai thuật toán là do cách chúng ta chọn phần tử x . Sẽ là tốt nếu chọn x để chia diện tích thành hai phần gần như bằng nhau, tức là. càng gần phần tử giữa càng tốt.

Làm thế nào để chọn một x như vậy? Chúng ta sẽ xem xét thuật toán được đề xuất bởi Bloom, Floyd, Pratt, Rivest và Tarjan. Hãy suy nghĩ theo cảm tính. Giả sử chúng ta biết một thuật toán tuyển tính

để tìm phần tử thứ k với $28n$ phép so sánh. Với $n \leq 55$, không khó để xác định một thuật toán cụ thể như vậy. Ví dụ, phương pháp bong bóng (xem 3.1.4.) Yêu cầu $n(n - 1)/2$ trong số phép so sánh để sắp xếp mảng hoàn chỉnh và đối với $n \leq 55$, áp dụng bất đẳng thức $28n \geq n(n - 1)/2$. Giả sử rằng phép so sánh $28t$ là đủ cho $t < n$. Chúng ta chia mảng thành $\lceil n/7 \rceil$ mảng con, mỗi mảng chứa 7 phần tử và nếu cần chúng ta bổ sung mảng con cuối cùng bằng $-\infty$ (chia), sau đó chúng ta sắp xếp hoàn chỉnh từng mảng con. Nếu chúng ta sử dụng phương pháp bong bóng, chúng ta sẽ cần không quá $7(7 - 1)/2 = 21$ phép so sánh. Như vậy, tổng số phép so sánh cần thiết không vượt quá $21(n/7) = 3n$. Sau đây là một ứng dụng đệ quy của thuật toán lựa chọn cho $n/7$ bảng được sắp xếp để tìm giá trị trung bình của các phương tiện (quy tắc). Điều này yêu cầu $28(n/7) = 4n$ phép so sánh.

Thuật toán tìm giá trị trung bình của các trung bình

1. Chia các phần tử thành các nhóm 7. Hãy ký hiệu chúng bằng S_i , với $i = 1, 2, \dots, \lceil n/7 \rceil$. Nhóm thứ hai có thể chứa ít hơn 7 phần tử.
2. Chúng ta sắp xếp hoàn toàn mỗi nhóm S_i và do đó chúng ta tìm thấy trung vị của nó m_i .
3. Chúng ta tìm trung vị M của các trung bình m_i bằng cách tham chiếu đệ quy đến cùng một thuật toán.

Chúng ta sẽ mô tả thuật toán trên kỹ hơn một chút với mã giả. Chúng ta sẽ xem xét rằng chúng ta có `partition()`, bởi một phần tử x cho trước chia mảng $L[]$ thành ba vùng: bên trái $L1[]$, trong đó các phần tử nhỏ hơn x , trung bình $L2[]$, trong đó chúng bằng x và $L3[]$, trong đó chúng lớn hơn x

Chia thành phần

```
unsigned select(int L[], /* Mảng */
               unsigned k) /* Đánh số phần tử tìm được*/
{
    n = length(L); /* Số lượng phần tử trong L[] */
    if (n <= 7) {
        bubbleSort(L,n);
        return L[k];
    }
    /* Chia L thành n/7 nhóp S[i] với đánh số 7 phần tử */
}
```

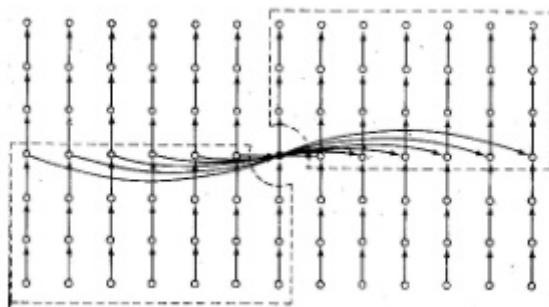
```

split(L, S = {S[i] | i = 1,2,...,n/7});
for (i = 0; i < n/7; i++)
    x[i] = select(S[i],3); /* Phần tử trung bình là thứ ba */
M = select({x[i] | i=1,2,...,n/7}, n/14);

partition(L,L1,L2,L3);

if (k <= length(L1)) then
    return select(L1,k);
else if (k > length(L1) + length(L2)) then
    return select(L3,k-length(L1)-length(L2));
else return M;
}

```



Hình 7.4. Chọn điểm trung vị

Trung vị được chọn của các phương tiện x đủ gần với phần tử trung bình, vì có ít nhất $2n/7$ nhỏ hơn x và ít nhất $2n/7$ lớn hơn x phần tử (xem Hình 7.4). Cho đến nay, chúng ta đã sử dụng 3n so sánh để sắp xếp các bảng, 4n khác - để tìm giá trị trung bình, n so sánh - để chia bảng theo phương pháp Hoor và không quá $28(5n/7)$ - so sánh cho ứng dụng đệ quy của thuật toán cho mỗi vùng tuần hoàn đệ quy giảm ít nhất $2n/7$ phần tử). Cuối cùng, hóa ra là chúng ta cần không quá $28n$ phép so sánh. Đó là, thuật toán của chúng ta là tuyến tính. Tất nhiên, 28 là một hằng số đủ nghiêm trọng, khác xa với 2, trong trường hợp tối ưu của các thuật toán trước đó. Hơn nữa, đối với $n \leq 16384$, thuật toán được đề xuất thực hiện nhiều phép so sánh hơn là sắp xếp hình chóp hoàn chỉnh của mảng. Tuy nhiên, tại $n > 16384$, thuật toán hoạt động đủ tốt trong

cả trường hợp chung và trường hợp xấu nhất. Ngoài ra, điểm số $28n$ có thể dễ dàng được hạ thấp hơn nữa, ví dụ như xuống còn $15n$. Bloom, Floyd, Pratt, Rivest và Tarjan quản lý để giảm nó xuống $391/72 \approx 5,4306.n$. Schönhager, Patterson và Pipenger ghi được $3n$, và Dor và Zwick đã tìm cách cải thiện số điểm đó lên $2,95n$ so với các phép so sánh! [Dor, Zwick-1996]

Bài tập

- ▷ 7.1. Hãy thực hiện một chương trình tìm phần tử lớn nhất thứ k , dựa trên việc sắp xếp bằng cách đếm.
- ▷ 7.2. Hãy chứng minh rằng trong thuật toán tìm phần tử lớn nhất thứ k dựa trên phép tách, tương tự như trong sắp xếp nhanh, không có vị trí tương hỗ nào khác của các biến i, j và k , ngoại trừ các vị trí được chỉ ra trong Hình 7.2, ?? và 7.3.
- ▷ 7.3. Hãy so sánh hai biến thể được đề xuất của thuật toán tìm phần tử lớn nhất thứ k , dựa trên phân tách, tương tự như sắp xếp nhanh.
- ▷ 7.4. Có nên mong đợi một cải tiến trong việc chia các phần tử thành các nhóm 5 khi tìm giá trị trung bình của các trung vị không? Và 3? Số lần so sánh dự kiến trong mỗi trường hợp là bao nhiêu? Xác định số phần tử tối ưu trong một nhóm.
- ▷ 7.5. Hãy thực hiện thuật toán tìm trung vị của các trung vị.

7.3. Phần tử trội

Định nghĩa 7.1. Giả sử là một đa tập hợp n phần tử (tức là một tập hợp trong đó sự lặp lại của các phần tử được phép). Chúng ta sẽ nói rằng một phần tử của tập hợp là *phần tử trội* nếu nó xuất hiện *nhiều hơn $n/2$ lần*.

Do đó, đa tập hợp $\{2, 3, 3, 1, 3\}$ có 3 là phần tử trội, còn đa tập hợp $\{1, 1, 2, 3\}$ không có phần tử trội.

Rõ ràng, theo định nghĩa trên, một tập hợp nhiều không thể có hai yếu tố phần tử trội. Dưới đây, chúng ta sẽ xem xét một số cách khác nhau để tìm ra nguyên nhân phần tử trội và chúng ta sẽ trình

bày tập hợp nhiều chiều với mảng một chiều. Trừ khi chúng ta đã tuyên bố rõ ràng khác, chúng ta sẽ không giả định bất kỳ loại phần tử cụ thể nào, cũng như sự tồn tại của một thứ tự (một phần hoặc toàn bộ) trong loại này, ít hơn nhiều là một thứ tự của các phần tử trong mảng. Trong các chương trình bên dưới, chúng ta sẽ sử dụng macro CDataType, mà chúng ta sẽ định nghĩa là char. Tính độc lập của thuật toán char có thể được kiểm tra bằng cách thay đổi macro, ví dụ thành int hoặc float.

Thuật toán đầu tiên có thể xuất hiện trong tâm trí người đọc là xem qua các phần tử của mảng và với mỗi phần tử để kiểm tra xem nó có xảy ra nhiều hơn $n/2$ lần hay không. Trong cách triển khai được đề xuất, hàm findMajority() được tìm thấy dưới dạng một tham số mảng, kích thước và địa chỉ của nó để lưu các majorant, trong trường hợp có một tham số trong mảng (hàm tác dụng phụ). Hàm trả về giá trị 1 nếu tìm thấy trường hợp phần tử trội và 0 - nếu không. Việc tìm số lần gặp của một phần tử trong một mảng được phân tách trong hàm count(), mà chúng ta sẽ sử dụng bên dưới cho các thuật toán khác. Trong trường hợp phát hiện sớm phần tử trội, công việc tiếp theo sẽ được chấm dứt ngay lập tức.

Chương trình 7.5. Tìm phần tử trội 1(705major1.c)

```
#include <stdio.h>
#define CDataType char

unsigned count(CDataType m[], unsigned size, CDataType candidate)
{
    unsigned cnt, i;
    for (i = cnt = 0; i < size; i++)
        if (m[i] == candidate)
            cnt++;
    return cnt;
}

char findMajority(CDataType m[], unsigned size, CDataType *majority)
{
    unsigned i, size2 = size / 2;
    for (i = 0; i < size; i++)
        if (count(m, size, m[i]) > size2) {
            *majority = m[i];
            return 1;
        }
}
```

```

        }
        return 0;
    }
int main() {
    CDataType majority;
    if (findMajority("AAACCBCCCCBCC", 13, &majority))
        printf("Phần tử trội là: %c\n", majority);
    else
        printf("Không có phần tử trội.\n");
    return 0;
}

```

Rõ ràng, độ phức tạp của thuật toán được đề xuất là $\Theta(n^2)$, vì mỗi phần tử trong số n phần tử được kiểm tra là phần tử trội, và kiểm tra yêu cầu một lần vượt qua khác. Chúng ta có thể "lừa" và giảm các lần kiểm tra mà không cần thay đổi đáng kể thuật toán. Vì mục đích này, cần lưu ý rằng tại mỗi lần gọi sau, hàm count() thực hiện ngày càng nhiều công việc thừa hơn, đi qua tất cả các mảng. Không khó để nhận thấy rằng việc đếm có thể dễ dàng bỏ sót các phần tử ở bên trái của phần tử hiện tại. Thật vậy, theo cách này, chúng ta có thể bỏ lỡ các lần gặp của phần tử được thử nghiệm. Tuy nhiên, trong trường hợp đó, nó sẽ không phải là phần tử trội, bởi vì trong một số bước trước đó, chúng ta lẽ ra đã xem xét nó và từ chối nó. Lập luận tương tự cho phép chúng ta hoàn thành việc xác minh $(n/2)$ phần tử của mảng. Thực vậy, nó là phần tử cuối cùng ở bên phải mà có đủ phần tử khác, có khả năng ngang hàng với nó, những phần tử có thể khiến chính nó trở thành phần tử trội.

Chương trình 7.6. Tìm phần tử trội cải tiến 2(706major2.c)

```

char findMajority(CDataType m[],unsigned size,CDataType *majority)
{
    unsigned i, j, cnt, size2 = size / 2;
    for (i = 0; i <= size / 2; i++) {
        for (cnt = 0, j = i; j < size; j++)
            if (m[i] == m[j]) cnt++;
        if (cnt > size2) {
            *majority = m[i];
            return 1;
        }
    }
}

```

```

    }
    return 0;
}

```

Thật không may, bất chấp những cải tiến được thực hiện, cả ở giữa và trong trường hợp xấu nhất, khi $n/2 + 1$ phần tử đầu tiên khác với nhau, độ phức tạp vẫn là $\Theta(n^2)$. Chúng ta có thể cố gắng hạn chế hơn nữa các yếu tố được xem xét trong quá trình đếm. Ý tưởng là ngừng đếm ngay khi thấy rõ rằng số lượng phần tử còn lại không đủ để làm cho phần tử được đề cập trở thành trường hợp phần tử trội, ngay cả khi tất cả chúng đều trùng khớp với nó.

Chương trình 7.7. Tìm phần tử trội cải tiến 3(707major3.c)

```

char findMajority(CDataType m[],unsigned size,CDataType *majority
)
{
    unsigned i, j, cnt, size2 = size / 2;
    for (i = 0; i <= size / 2; i++) {
        for (cnt = 0, j = i; j < size; j++)
            if (cnt + size - j <= size2)
                break;
            else if (m[i] == m[j])
                cnt++;
        if (cnt > size2) {
            *majority = m[i];
            return 1;
        }
    }
    return 0;
}

```

Tuy nhiên, lợi ích của "cải tiến" này trong trường hợp chung là khá đáng ngờ, vì nó yêu cầu bổ sung: một phép cộng, một phép trừ và một phép so sánh ở mỗi bước của số đếm. Có nghĩa là, trong trường hợp xấu nhất, quyết định này tồi tệ hơn quyết định trước. Mặc dù độ phức tạp tiệm cận của nó lại là $\Theta(n^2)$.

Chúng ta sẽ không nghiên cứu sâu hơn theo hướng này, vì có nhiều thuật toán hiệu quả hơn. Hãy xem điều gì sẽ xảy ra nếu chúng ta làm suy yếu các hạn chế nghiêm ngặt được áp đặt ngay từ đầu và cho phép loại yếu tố mà một quy định hoàn chỉnh được xác định.

Trong trường hợp này, chúng ta có thể sắp xếp các phần tử của mảng và kiểm tra xem phần tử ở giữa của nó có phải là trường hợp phần tử trội hay không. Chúng ta nhận được thuật toán sau (Bằng chứng rằng nếu chúng ta sắp xếp mảng, phần tử ở giữa của nó sẽ bị chiếm bởi phần tử phần tử trội, nếu có, chúng ta để nó cho người đọc như một bài tập nhẹ.)

Chương trình 7.8. Tìm phần tử trội cải tiến 4(708major4.c)

```
char findMajority(CDataType m[],unsigned size,CDataType *majority)
{
    heapSort(m, size); /* hoặc mergeSort(m,size); */
    if (count(m, size, m[size / 2]) > size/2) {
        *majority = m[size / 2];
        return 1;
    }
    return 0;
}
```

Bởi vì việc kiểm tra xem một phần tử có phải là phần tử trội hay không là tuyến tính, độ phức tạp của thuật toán bị chi phối bởi độ phức tạp của sắp xếp. Nếu chúng ta sử dụng một thuật toán như sắp xếp theo hình chóp (xem ??) Hoặc sắp xếp hợp nhất (xem 7.5. Bên dưới), có độ phức tạp đảm bảo $\Theta(n \log_2 n)$ trong trường hợp xấu nhất, chúng ta nhận được độ phức tạp tổng thể là thuật toán $\Theta(n \log_2 n)$. Chúng ta sẽ lưu ý rằng việc phân loại nhanh Hoor sẽ không hiệu quả với chúng ta. Mặc dù trong trường hợp trung bình, nó có độ phức tạp là $\Theta(n \log_2 n)$ và chia ra từ 2 đến 3 lần, trong trường hợp xấu nhất thì độ phức tạp của nó là $\Theta(n^2)$.

Tuy nhiên, để tìm phần tử ở giữa (và lớn nhất thứ k) của một mảng, không cần thiết phải sắp xếp nó. Nhớ lại rằng thuật toán của Bloom, Floyd, Pratt, Rivest và Tarjan tìm phần tử trung bình của độ phức tạp tuyến tính $\Theta(n)$, tạo ra $5,43n - 163$ phép so sánh, $n > 32$ (xem 7.2)

Chương trình 7.9. Tìm phần tử trội cải tiến 5(709major5.c)

```
char findMajority(CDataType m[],unsigned size,CDataType *majority)
{
    CDataType med;
    med = findMedian(m, size);
```

```

if (count(m, size, med) > size2) {
    *majority = med;
    return 1;
}
return 0;
}

```

Do đó, chúng ta đã thu được một thuật toán tuyến tính để tìm một trường hợp phần tử trội, nhưng với giả thiết bổ sung rằng có một thứ tự tuyến tính giữa các phần tử. Hãy xem những gì chúng ta sẽ nhận được nếu chúng ta từ bỏ thứ tự, nhưng yêu cầu số lượng các giá trị khác nhau có thể có của các phần tử của mảng phải được biết trước và là một số đủ nhỏ.

Trong trường hợp này, chúng ta lại có thể nhận được một thuật toán tuyến tính. Về bản chất, ý tưởng của thuật toán đếm được sử dụng (xem ??). Với một lần đi qua các phần tử của mảng là số cuộc họp của tất cả các ứng viên cho trường hợp phần tử trội. Tiếp theo là xem qua các giá trị có thể có của các ứng viên và kiểm tra từng giá trị xem nó có xảy ra đúng hơn $n/2$ lần hay không. Điều này xảy ra với thời gian $\Theta(k)$, không phụ thuộc vào số lượng phần tử trong mảng mà chỉ dựa trên số lượng các giá trị khác nhau của k mà chúng có thể chấp nhận. Do đó, với tổng độ phức tạp của thuật toán, chúng ta nhận được $\Theta(n + k)$. Với các giá trị đủ nhỏ của k (ví dụ $k < n$) ta có $\Theta(n)$.

Chương trình 7.10. Tìm phần tử trội cải tiến 6 (710major6.c)

```

#define MAX_NUM 127
CDataType cnt[MAX_NUM + 1];
char findMajority(CDataType m[],unsigned size,CDataType *majority)
{
    unsigned i, j, size2 = size / 2;
    /* Khởi tạo */
    for (i = 0; i < MAX_NUM; i++)
        cnt[i] = 0;
    /* đếm */
    for (j = 0; j < size; j++)
        cnt[m[j]]++;
    /* Kiểm tra trội */
    for (i = 0; i < MAX_NUM; i++)

```

```

if (cnt[i] > size2) {
    *majority = i;
    return 1;
}
return 0;
}

```

Tuy nhiên, chúng ta hãy quay lại điều kiện ban đầu, trong đó chúng ta không thể dựa vào sắc lệnh cũng như một số lượng nhỏ các giá trị có thể có của các phần tử đã biết trước, và cố gắng thực hiện chiến lược chia để trị. Chia mảng thành hai phần gần như bằng nhau (trong trường hợp số phần tử lẻ thì một phần sẽ có thêm một phần tử). Ý tưởng cơ bản là nếu x là điều kiện phần tử trội của mảng đầu ra, thì nó sẽ là điều kiện phần tử trội của ít nhất một trong hai phần. Từ đây ta dễ dàng có được thuật toán đệ quy tương ứng. Chúng ta chia mảng thành hai phần gần như bằng nhau và tìm kinh giới trên mỗi phần. Có ba trường hợp:

1) Cả hai mảng con đều không có yếu tố phần tử trội. Khi đó sẽ không có mảng đầu ra.

2) Một mảng con có phần tử trội x và mảng còn lại thì không. Kiểm tra xem x có phải là phần tử trội của mảng đầu ra hay không.

3) Cả hai mảng con đều có các đại lượng x và y , có thể khác nhau. Cả x và y đều được kiểm tra. (Nếu x là một trường hợp phần tử trội, thì bài kiểm tra y có thể bị bỏ qua.)

Thuật toán tương tự được áp dụng một cách đệ quy để tìm điều phần tử trội của từng mảng con và quá trình kết thúc khi đến một mảng có một phần tử, trong trường hợp đó phần tử này là trường hợp phần tử trội. Quá trình có thể kết thúc sớm hơn (ví dụ: với hai yếu tố) vì lý do hiệu quả, nhưng chúng ta sẽ không xem xét kỹ lưỡng theo hướng này. Chúng ta sẽ lưu ý rằng việc duy trì các giới hạn bên trái và bên phải yêu cầu một sự thay đổi nhỏ trong các tham số của cả hai hàm; lệnh gọi ban đầu được thực hiện với biên bên trái của mảng được coi là 0 và bên phải - size - 1

Chương trình 7.11. Tìm phần tử trội cải tiến 7 (711major7.c)

```

unsigned count(CDataType m[], unsigned left,
                unsigned right, CDataType candidate)

```

```

{ unsigned cnt;
  for (cnt = 0; left <= right; left++)
    if (m[left] == candidate)
      cnt++;
  return cnt;
}

char findMajority(CDataType m[], unsigned left,
                  unsigned right, CDataType *majority)
{ unsigned mid;
  if (left == right) {
    *majority = m[left];
    return 1;
  }
  mid = (left + right) / 2;
  if (findMajority(m, left, mid, majority))
    if (count(m, left, right, *majority) > (right - left + 1)/2)
      return 1;
  if (findMajority(m, mid + 1, right, majority))
    if (count(m, left, right, *majority) > (right - left + 1)/2)
      return 1;
  return 0;
}

```

Chúng ta đã đạt được những gì? Trong trường hợp tổng quát, ở mỗi mức đệ quy, chúng ta có hai phép nghịch lưu đệ quy với mảng, với ít phần tử hơn một nửa. Đây là phần tách. Kết hợp các kết quả của hai phép nghịch lưu đệ quy (phần quy tắc) yêu cầu các phép so sánh 0, n hoặc $2n$, tương ứng cho các trường hợp 1), 2) và 3). Trong trường hợp giới hạn của một phần tử, chúng ta có độ phức tạp không đổi mà không cần so sánh. Nếu chúng ta biểu thị số lượng phép so sánh được thực hiện cho một mảng n phần tử bằng $T(n)$, trong trường hợp xấu nhất, chúng ta thu được các phụ thuộc lặp lại sau:

$$T(1) = 0$$

$$T(n) = T(n/2) + 2n$$

Sử dụng định lý cơ bản (xem 1.4.10) chúng ta thu được rằng $T(n) \in \Theta(n \log_2 n)$. Chúng ta không thể đạt được kết quả tốt hơn

sao? Hóa ra là chúng ta có thể dễ dàng thu được độ phức tạp tuyến tính của thuật toán, lưu ý rằng nếu mảng có một trường hợp phần tử trội, thì các câu lệnh sau là hợp lệ:

Mệnh đề 7.1. *Nếu mảng có một phần tử phần tử trội và chúng ta loại bỏ hai phần tử khác nhau, phần tử đặc biệt của mảng mới sẽ trùng với phần tử gốc.*

Mệnh đề 7.2. *Nếu tất cả các phần tử trong một mảng gấp nhau thành từng cặp, thì nếu chúng ta giữ một đại diện và loại bỏ cái kia cho mỗi cặp, thì phần tử phần tử trội của mảng mới sẽ trùng với phần tử phần tử trội của mảng ban đầu.*

Chứng minh cho những nhận định trên là nhẹ và chúng ta xin để lại cho bạn đọc. Chúng ta sẽ lưu ý rằng khi xóa hai phần tử khác nhau khỏi một mảng mà không có chức năng phần tử trội, chúng ta có thể nhận được một mảng có chức năng phần tử trội. Ví dụ: nếu chúng ta loại bỏ 3 và 4 (khác nhau) khỏi mảng $\{1, 1, 2, 3, 4\}$ (không chứa phần tử trội), chúng ta nhận được mảng $\{1, 1, 2\}$, trong đó 1 là phần tử trội.

Hãy thử đơn giản hóa điều kiện một lần nữa bằng cách yêu cầu: 1) mảng được đảm bảo chứa một điều kiện phần tử trội và 2) cho phép thay đổi mảng để sau khi chương trình kết thúc, nó (gần như chắc chắn) khác với ban đầu. Hãy giả sử một chút rằng mảng có một số phần tử chẵn. Hãy thử lại để áp dụng chiến lược La Mã, chia mảng thành hai phần bằng nhau: phần 1 - các phần tử có vị trí lẻ và phần 2 - các phần tử của vị trí chẵn. Tuy nhiên, thay vì tìm kiếm điểm phần tử trội trong mỗi chúng, lần này chúng ta sẽ so sánh các yếu tố theo từng cặp: một yếu tố từ bộ phận này và yếu tố kia từ bộ phận kia. Chúng ta xem xét một cặp cụ thể (các phần tử liên tiếp). Nếu hai yếu tố khác nhau, chúng ta loại trừ chúng khỏi việc xem xét thêm và nếu chúng bằng nhau - chúng ta giữ nguyên một yếu tố. Áp dụng điều này cho tất cả các cặp, chúng ta nhận được một mảng mới có nhiều nhất $n/2$ phần tử. Chiến lược tương tự cũng được áp dụng cho anh ta, vì quá trình này tiếp tục cho đến khi thu được một mảng có một phần tử: phần tử phần tử trội (chúng ta nhắc bạn rằng chúng ta giả định rằng chắc chắn có một phần tử trội).

Sự cố xảy ra với một số phần tử lẻ, trong đó một trong các phần tử sẽ không thể được đưa vào một cặp. Vì yếu tố này có thể quan trọng và xác định yếu tố phần tử trội, chúng ta sẽ không muốn vứt bỏ nó. (Ví dụ: AA BB A). Mặt khác, chúng ta không muốn luôn giữ nó, vì bằng cách đó chúng ta có thể phá hủy tính chất phần tử trội (Ví dụ: AA B).

Một giải pháp khả thi là giữ nguyên, giảm trọng lượng và chỉ tính đến nếu trong mảng mới không có nó thì không thể xác định được trường hợp phần tử trội. Chúng ta nhận được một thuật toán có độ phức tạp tuyến tính, bởi vì ở mỗi bước số lượng phần tử giảm ít nhất một nửa. Nếu chúng ta biểu thị số lượng phép so sánh được thực hiện cho một mảng n phần tử bằng $T(n)$, trong trường hợp xấu nhất, chúng ta thu được các phụ thuộc lặp lại sau:

$$T(1) = 0$$

$$T(n) = T(n/2) + n/2$$

Sử dụng định lý cơ bản (xem 1.4.10.) Ta thu được $T(n) \in \Theta(n)$. Việc thực hiện chương trình được đề xuất không sử dụng một mảng bổ sung, nhưng ở mỗi bước sao chép liên tiếp một đại diện của các phần tử của các cặp có cùng phần tử ở đầu cùng một mảng. Dễ dàng nhận thấy rằng điều này không hề nguy hiểm, vì việc sao chép diễn ra “sau lưng”. Thật không may, điều này phá hủy mảng ban đầu, làm mất đi cơ hội của chúng ta để kiểm tra xem phần tử được tìm thấy có thực sự là một nguyên nhân phần tử trội hay không.

Chương trình 7.12. Tìm phần tử trội cải tiến 8 (712major8.c)

```
void findMajority(CDataType m[], unsigned size, CDataType *majority)
{
    unsigned i, curCnt;
    char part = 0;
    do {
        for (curCnt = 0, i = 1; i < size; i += 2)
            if (m[i - 1] == m[i])
                m[curCnt++] = m[i];
        if (i == size) {
            m[curCnt++] = m[i - 1];
            part = 1;
        }
        else if (part)
```

```

m[curCnt] = m[size - 2];
else if (m[size - 2] == m[size - 1])
    m[curCnt] = m[size - 2];
else
    curCnt--;
size = curCnt;
} while (size > 1);
*majority = m[0];
}

```

Một khả năng khác là lưu phần tử cuối cùng của mảng với một số phần tử lẻ trong một biến riêng biệt đặc biệt. Nếu trong bước tiếp theo, mảng có một số phần tử lẻ, chúng ta sẽ hủy nó bằng cách viết phần tử cuối cùng mới của mảng lên trên nó. Tại một thời điểm, mảng không có phần tử nào và ứng cử viên cho trường hợp phần tử trội sẽ nằm trong biến đặc biệt. Điều này có hiệu quả không? Sang bước tiếp theo, chúng ta sẽ chỉ mất đi phần tử majorant nếu trong mảng mới có đúng một nửa số phần tử bằng với phần tử majorant và phần tử dành riêng của chúng ta nhất thiết phải bằng phần tử majorant (chúng ta biết rằng chắc chắn có một phần tử phần tử trội trong mảng). Trong mỗi bước tiếp theo, ít nhất một nửa số phần tử sẽ bằng với phần tử trội, vì vậy nếu tại một thời điểm chúng ta thấy mình có một số phần tử lẻ, phần tử còn lại sẽ lại phải bằng phần tử trội, điều đó có nghĩa là chúng ta có thể loại bỏ giá trị cũ một cách an toàn. Sau khi các phần tử của mảng hết, kinh giới sẽ nằm trong biến đặc biệt. Số phần tử lại giảm ít nhất một nửa ở mỗi bước, tức là chúng ta có độ phức tạp tuyến tính.

Chương trình 7.13. Tìm phần tử trội cải tiến 9 (713major9.c)

```

void findMajority(CDataType m[], unsigned size, CDataType *majority)
{
    unsigned i, curCnt;
    do {
        for (curCnt = 0, i = 1; i < size; i += 2)
            if (m[i - 1] == m[i])
                m[curCnt++] = m[i];
        if (size & 1)
            *majority = m[size - 1];
        size = curCnt;
    }
}

```

```

    } while (size > 0);
}

```

Chúng ta không thể loại bỏ sự cần thiết của một yếu tố bổ sung? Hãy nghĩ xem khi nào chúng ta cần xem xét yếu tố cuối cùng. Không khó để coi rằng nó sẽ không cần thiết nếu số phần tử trong mảng mới là số lẻ, vì trong trường hợp đó không thể có hai ứng cử viên cho trường hợp phần tử trội. Tuy nhiên, nếu con số này là số chẵn, thì có thể không xác định được điều kiện phần tử trội (tức là các phần tử chứa hai giá trị được phân bổ bằng nhau), đó là lý do tại sao anh ta sẽ xác định điều kiện phần tử trội. Vì vậy: trong trường hợp một số phần tử lẻ, phần tử cuối cùng chỉ được thêm vào cuối mảng mới nếu không thì số phần tử của nó sẽ trở thành số chẵn. Do đó, khi chúng ta đến một mảng có số phần tử lẻ, chúng ta sẽ lưu trữ một số lẻ cho đến khi chỉ còn lại một phần tử duy nhất trong đó: majorant. Thuật toán được mô tả là một biến thể đơn giản của thuật toán trước và độ phức tạp của nó lại là tuyến tính: Nếu cần, phần tử cuối cùng của mảng sẽ tiếp nhận các chức năng của biến đặc biệt từ thuật toán trước đó.

Chương trình 7.14. Tìm phần tử trội cải tiến 10 (714major10.c)

```

void findMajority(CDataType m[],unsigned size,CDataType *majority
)
{
    unsigned i, curCnt;
    do {
        for (curCnt = 0, i = 1; i < size; i += 2)
            if (m[i - 1] == m[i])
                m[curCnt++] = m[i];
        if (!(curCnt & 1))
            m[curCnt++] = m[size - 1];
        size = curCnt;
    } while (size > 1);
    *majority = m[0];
}

```

Một cách khác có thể (nhưng phức tạp hơn không cần thiết) để giải bài toán về số phần tử lẻ là liên kết một bộ đếm với mỗi phần tử, cho biết nó thực sự đại diện cho bao nhiêu phần tử. Ban đầu, các bộ đếm của tất cả các phần tử được khởi tạo bằng 1. Khi so sánh

hai phần tử $m[i-1]$ và $m[i]$, với các bộ đếm lần lượt là $cnt[i-1]$ và $cnt[i]$, chúng ta có hai lựa chọn:

1. $m[i-1] == m[i]$

Chúng ta lưu một bản sao của phần tử, với bộ đếm $cnt[i-1] + cnt[i]$.

2. $m[i-1] != m[i]$

2.1. $cnt[i-1] == cnt[i]$

Chúng ta loại bỏ cả hai yếu tố.

2.2. $cnt[i-1] < cnt[i]$

Ta lưu $m[i]$ với bộ đếm $cnt[i] - cnt[i-1]$.

2.3. $cnt[i-1] > cnt[i]$

Ta lưu $m[i-1]$ với bộ đếm $cnt[i-1] - cnt[i]$.

Bây giờ chúng ta luôn có thể lưu phần tử cuối cùng trong trường hợp một số phần tử lẻ. Đối với trường hợp là số chẵn thì tất nhiên chúng ta sẽ không giữ lại, vì nó sẽ được gộp thành từng cặp. Để dàng nhận thấy rằng thuật toán kết quả lại là tuyến tính và ở mỗi bước, điều kiện phần tử trội được giữ nguyên. Sau đây là một ví dụ triển khai:

Chương trình 7.15. Tìm phần tử trội cải tiến 11 (715major11.c)

```
void findMajority(CDataType m[], unsigned size, CDataType *majority)
{
    unsigned i, curCnt;
    unsigned *cnt = (unsigned *) malloc(size * sizeof(*cnt));
    for (i = 0; i < size; i++) cnt[i] = 1;
    do {
        for (curCnt = 0, i = 1; i < size; i += 2) {
            if (m[i - 1] == m[i]) {
                cnt[curCnt] = cnt[i - 1] + cnt[i];
                m[curCnt++] = m[i];
            }
            else if (cnt[i] > cnt[i - 1]) {
                cnt[curCnt] = cnt[i] - cnt[i - 1];
                m[curCnt++] = m[i];
            }
            else if (cnt[i] < cnt[i - 1]) {
                cnt[curCnt] = cnt[i - 1] - cnt[i];
                m[curCnt++] = m[i - 1];
            }
        }
    }
```

```

    }
    if (size & 1) {
        cnt[curCnt] = cnt[i - 1];
        m[curCnt++] = m[i - 1];
    }
    size = curCnt;
} while (size > 1);
free(cnt);
*majority = m[0];
}

```

Một giải pháp thay thế để giải quyết vấn đề thời gian $\Theta(n)$ là sử dụng một ngăn xếp (xem ??). Lần này chúng ta sẽ loại bỏ yêu cầu mảng phải chứa cây kinh giới. Chúng ta bắt đầu với một ngăn xếp trống, trong đó phần tử đầu tiên của mảng nhập vào ở đầu. Sau đó, ở mỗi bước của mảng, phần tử tiếp theo được trích xuất và so sánh với phần tử ở trên cùng của ngăn xếp. Nếu các phần tử giống nhau, phần tử mới sẽ đi vào ngăn xếp. Nếu không, phần tử trên cùng ngăn xếp bị loại trừ, điều này thực tế có nghĩa là từ chối cả hai phần tử. Trong trường hợp ngăn xếp trống, phần tử tiếp theo sẽ nhập vào đó. Quá trình tiếp tục cho đến khi hết các phần tử của mảng. Nếu có điều gì phần tử trội, nó ở trên cùng của ngăn xếp. Thực vậy, các phần tử bị phá hủy đồng thời theo từng cặp, và chỉ khi chúng khác nhau. Do đó, không thể hủy phần tử trội vì không có đủ các phần tử không phải phần tử trội trong mảng. Thực vậy, theo định nghĩa, điều kiện phần tử trội hoàn toàn *bằng hơn* một nửa số phần tử.

Tuy nhiên, mảng có thể không chứa phần tử phần tử trội và vẫn có một phần tử ở đầu ngăn xếp, ví dụ:

- "ABC" (ngăn xếp: \emptyset)
- "BC" (ngăn xếp: A)
- "C" (ngăn xếp: \emptyset)
- " " (Ngăn xếp: C)

Điều này yêu cầu một đường truyền bổ sung qua mảng để kiểm tra xem phần tử có thực sự là vật phần tử trội hay không. Lưu ý rằng trong trường hợp này, điều này có thể xảy ra vì mảng ban đầu không bị phá hủy, nhưng điều này gây tốn kém bộ nhớ ngăn xếp bổ sung. Lưu ý rằng ngăn xếp có thể chứa nhiều hơn một phần tử.

Trong trường hợp này, tất cả các phần tử trong nó sẽ có cùng giá trị, theo lý luận ở trên. Sau đây là một ví dụ triển khai thuật toán:

Chương trình 7.16. Tìm phần tử trội cài tiến 12 (716major12.c)

```
/* Biến, Hàm và định nghĩa ngăn xếp*/
#define STACK_SIZE 100
CDataType stack[STACK_SIZE];
unsigned stIndex;
void stackInit(void) { stIndex = 0; }
void stackPush(CDataType elem) { stack[stIndex++] = elem; }
CDataType stackPop(void) { return stack[--stIndex]; }
CDataType stackTop(void) { return stack[stIndex - 1]; }
char stackIsEmpty(void) { return 0 == stIndex; }

char findMajority(CDataType m[], unsigned size, CDataType *majority)
{
    unsigned i, cnt;
    stackInit();
    for (stackPush(m[0]), i = 1; i < size; i++) {
        if (stackIsEmpty())
            stackPush(m[i]);
        else if (stackTop() == m[i])
            stackPush(m[i]);
        else
            stackPop();
    }
    if (stackIsEmpty()) return 0;
    for (*majority = stackPop(), i = cnt = 0; i < size; i++)
        if (m[i] == *majority)
            cnt++;
    return(cnt > size / 2);
}
```

Chúng ta có thực sự cần một ngăn xếp không? Không khó để thấy rằng trong thuật toán đề xuất ngăn xếp luôn chứa các phần tử có cùng giá trị. Thật vậy, nếu phần tử hiện tại khác với phần tử ở trên cùng của ngăn xếp (giả sử ngăn xếp không trống), thì nó không thể được thêm vào, mà thay vào đó, phần tử ở trên cùng của ngăn xếp sẽ hủy lẫn nhau. Nhưng sau đó chúng ta có thể loại bỏ ngăn xếp và thay thế nó bằng một cặp biến như: (phần tử hiện tại, bộ đếm).

Lúc đầu, ứng cử viên có giá trị không xác định và bộ đếm có giá trị bằng 0. Thuật toán chuyển tuần tự qua các phần tử của mảng và ở mỗi bước thực hiện kiểm tra và các hành động liên sau:

1) Nếu bộ đếm là 0, phần tử hiện tại trở thành một ứng cử viên và bộ đếm trở thành 1.

2) Nếu bộ đếm khác 0:

2.1) Nếu ứng cử viên phù hợp với phần tử hiện tại, bộ đếm sẽ tăng lên 1.

2.2) Nếu khác, bộ đếm giảm đi 1.

Quá trình tiếp tục cho đến khi hết các phần tử của mảng. Nếu cuối cùng bộ đếm trở thành 0, mảng chắc chắn không chứa phần tử trội. Tuy nhiên, nếu nó khác 0, điều đó không có nghĩa là ứng viên đó nhất thiết phải là một trường hợp phần tử trội và cần được kiểm tra tương ứng. Một khác, nếu mảng có điều kiện phần tử trội, thì điều này nhất thiết phải là ứng cử viên.

Sau đây là một ví dụ về việc áp dụng thuật toán cho một chuỗi cụ thể có chứa điều kiện khó hiểu.

A A A C C B B C C C B C C

^

? : 0

A A A C C B B C C C B C C

^

A : 1

A A A C C B B C C C B C C

^

A : 2

A A A C C B B C C C B C C

^

A : 3

A A A C C B B C C C B C C

^

A : 2

A A A C C B B C C C B C C

^

A : 1

A A A C C B B C C C B C C

^

? : 0

```

A A A C C B B C C C B C C
^

B:1
A A A C C B B C C C B C C
^

?:0
A A A C C B B C C C B C C
^

C:1
A A A C C B B C C C B C C
^

C:2
A A A C C B B C C C B C C
^

C:1
A A A C C B B C C C B C C
^

C:2
A A A C C B B C C C B C C
^

C:3

```

Trong trường hợp của chúng ta, điều kiện phần tử trội là C và thuật toán xác định chính xác anh ta là một ứng cử viên. Lưu ý rằng nếu chúng ta thay thế, ví dụ: C đầu tiên bằng A, thuật toán sẽ lại chỉ ra C là ứng cử viên cho trường hợp phần tử trội, nhưng trong trường hợp này đơn giản là không có trường hợp phần tử trội. Đó là, cần phải có một cuộc kiểm tra cuối cùng tiêu chuẩn để đảm bảo rằng ứng viên được đề cử trên thực tế là một trường hợp phần tử trội. Sau đây là một ví dụ triển khai:

Chương trình 7.17. Tìm phần tử trội cải tiến 13 (717major13.c)

```

char FindMajority(CDataType m[],unsigned size,CDataType *
                     majority)
{ unsigned cnt, i;
  for (i = cnt = 0; i < size; i++) {
    if (0 == cnt) {
      *majority = m[i];
      cnt = 1;
    }
  }
}

```

```

        else if (m[i] == *majority) cnt++;
        else cnt--;
    }
    if (cnt > 0) {
        for (i = cnt = 0; i < size; i++)
            if (m[i] == *majority)
                cnt++;
        return(cnt > size / 2);
    }
    return 0;
}

```

Lưu ý rằng thuật toán này, không giống như một số thuật toán trước, sử dụng bộ nhớ bổ sung không đổi (cho hai biến) và không thay đổi mảng. Mặc dù nó có thể được coi là một biến thể của cái trước, nhưng chúng ta sẽ cố gắng giải thích và biện minh cho nó từ một quan điểm khác.

Giả sử thuật toán đã bắt đầu và đạt đến một vị trí. Chúng ta có thể giả định rằng anh ta đã chia các phần tử được kiểm tra của mảng thành hai khu vực. Một khu vực chứa các phần tử được nhóm theo cặp, vì vậy các thành viên của mỗi cặp là khác nhau. Các yếu tố khác bằng nhau và ngang bằng với ứng viên chuyên ngành. Có thể dễ dàng nhận thấy rằng nếu mảng chứa điều kiện phần tử trội, thì đại diện của nó chắc chắn sẽ rơi vào nhóm thứ hai. Thực vậy, điều ngược lại là không thể, vì không có đủ các yếu tố khác ngoài yếu tố phần tử trội để kết đôi với anh ta, vì vậy tất cả các cuộc họp của phần tử trội đều rơi vào nhóm đầu tiên. Tuy nhiên, nếu mảng không có phần tử trội, có thể nhóm thứ hai vẫn bao gồm phần tử ứng viên (ví dụ: ABC, C sẽ là một ứng viên). Đó là lý do tại sao ở đây cũng cần có lối đi bổ sung quen thuộc qua khói núi. Nếu nhóm thứ hai trở nên trống rỗng, thì không có gì đảm bảo về phần tử trội trong mảng. Chúng ta sẽ lưu ý rằng mặc dù một số thuật toán tuyến tính đã được xem xét, nhưng các hằng số ẩn không được quên. Ví dụ: thuật toán thứ hai tốt hơn thuật toán của Bloom, Floyd, Pratt, Rivest và Tarjan để tìm giá trị trung bình, hoạt động dưới những ràng buộc chặt chẽ hơn. Thực vậy, ở đây số phép so sánh là $2n$, trong khi trong thuật toán của Bloom, Floyd, Pratt, Rivest và Tarjan là $5,43n - 163$, $n > 32$. Muốn vậy chúng ta phải thêm $n - 1$ phép so

sánh để kiểm tra xem trung vị có phải là phần tử trội, trong đó chúng ta nhận được $6,43n - 164, n > 32$ (xem 7.1.)

Bài tập

- ▷ 7.6. Chứng minh rằng nếu một mảng có chứa phần tử trội được sắp xếp, thì phần tử ở giữa của nó bị chiếm bởi phần tử trội.
- ▷ 7.7. Chứng minh rằng nếu x là trường hợp phần tử trội của một mảng và mảng được chia thành hai phần gần như bằng nhau thì x là trường hợp phần tử trội của ít nhất một phần trong số chúng.
- ▷ 7.8. Chứng minh rằng nếu mảng $m[]$ có một giá trị phần tử trội, thì nếu các phần tử $m[i]$ và $m[j]$ khác nhau và bị loại bỏ, thì giá trị phần tử trội của mảng mới sẽ trùng với giá trị phần tử trội của mảng cũ.
- ▷ 7.9. Chứng minh rằng nếu một mảng có một phần tử phần tử trội, thì nếu tất cả các phần tử trong mảng gặp nhau thành từng cặp và nếu một đại diện được giữ lại bằng cách loại bỏ cái kia cho mỗi cặp thì phần tử phần tử trội của mảng mới sẽ trùng với phần tử phần tử trội của mảng cũ.

7.4. Hợp nhất các mảng đã sắp xếp

Cho hai dãy đã sắp xếp A và B là các miền con của mảng $a[]$. bài toán là kết hợp chúng theo một cách thích hợp để chuỗi kết quả C cũng được sắp xếp. Thoạt nhìn, có vẻ như điều này có thể được thực hiện ngay tại chỗ, tức là trực tiếp trong mảng $a[]$. Chúng ta để người đọc tự thấy rằng điều này không dễ dàng như vậy. Trên thực tế, một phương pháp như vậy có tồn tại, nhưng nó khá phức tạp và đòi hỏi sự chú ý đặc biệt và nỗ lực thêm. Thay vào đó, ở đây chúng ta sẽ đề xuất thuật toán cổ điển với một mảng bổ sung $b[]$, trong đó các phần tử của $a[]$ ban đầu được sao chép, sau đó hai phân vùng hợp nhất thành một $a[]$.

Trước tiên, chúng ta hãy xem xét trường hợp tổng quát hơn của việc hợp nhất hai mảng đã sắp xếp $a[]$ và $b[]$ (với n và m phần tử tương ứng) thành một mảng được sắp xếp mới $c[]$. Điều này có thể được thực hiện như sau: Chúng ta khởi tạo 3 chỉ mục: một cho mỗi

mảng. Ta so sánh các phần tử đầu tiên của hai mảng $a[]$ và $b[]$ rồi chuyển phần nhỏ hơn thành $c[]$, sau đó tăng các chỉ số của $c[]$ và mảng chứa phần tử nhỏ hơn. Trong bước tiếp theo, chúng ta lại so sánh các phần tử tương ứng của hai mảng và sao chép mảng nhỏ hơn trong $c[]$, tăng các chỉ số tương ứng. Quá trình kết thúc khi một trong các mảng bị cạn kiệt, sau đó các phần tử khác của mảng chưa được xử lý sẽ được sao chép vào $c[]$. Sau đây là một ví dụ triển khai thuật toán được mô tả (hai vòng lặp while cuối cùng có thể được thay thế bằng `memcpy()`):

Thay vòng lặp chu kỳ lồng nhau

```
i = j = k = 0;
while (i < n && j < m)
    c[k++] = (a[i] < b[j]) ? a[i++] : b[j++];
if (i == n)
    while(j < m)
        c[k++] = b[j++];
else
    while(i < n) c[k++] = a[i++];
```

Chúng ta có thể cố gắng tối ưu hóa phân đoạn chương trình trên bằng cách giảm số lượng so sánh được thực hiện trong chu kỳ trong `while` đầu tiên. Điều này có thể đạt được với sự trợ giúp của bộ giới hạn (phần tử dừng bổ sung ∞) trong mảng đầu ra $a[]$ và $b[]$, do đó không thể đi qua các cạnh của chúng. Bất lợi của việc triển khai là các so sánh dư thừa được thực hiện khi các phần tử của $a[]$ hoặc $b[]$ đã cạn kiệt

```
i = j = k = 0; a[m] = b[n] = INFINITY; mn = m + n;
while (k < mn)
    c[k++] = (a[i] < b[j]) ? a[i++] : b[j++];
```

Thuật toán mới nhất thiết yêu cầu chính xác $n + m$ phép so sánh, trong khi thuật toán đầu tiên yêu cầu ít hơn nếu sử dụng hàm `memcpy()`. Trên thực tế, hàm `memcpy()` cũng được triển khai nội bộ thông qua một vòng lặp được liên kết với các phép so sánh có liên quan, nhưng việc triển khai nó khá hiệu quả, vì vậy có lẽ tùy chọn sau là phù hợp hơn (tuy nhiên, có một so sánh bổ sung và một số bộ sưu tập ở đây):

Sửa đổi hàm memcpy()

```
i = j = k = 0;
while (i < n && j < m)
    c[k++] = (a[i] < b[j]) ? a[i++] : b[j++];
if (i == n)
    memcpy(c+k,b+j,m-j);
else
    memcpy(c+k,a+i,n-i);
```

Thuật toán được mô tả được triển khai tốt nhất bằng cách sử dụng cấu trúc dữ liệu động, vì việc chuyển một phần tử từ A hoặc B sang C được thực hiện đơn giản bằng cách chuyển hướng con trỏ, tức là không cần thêm bộ nhớ cho C và khi hết các phần tử của một chuỗi, các phần tử của phần tử kia được dán một cách tầm thường vào cuối C. Tuy nhiên, mọi thứ không quá rõ rỡ, vì các con trỏ yêu cầu thêm bộ nhớ và kết quả là hoạt động của thuật toán A và B bị phá hủy.

Thuật toán trên có thể được tổng quát hóa cho bất kỳ số lượng trình tự nào, mỗi chuỗi duy trì con trỏ chỉ mục của riêng nó với một chức năng tương tự như i và j của việc triển khai ở trên. Chương trình được đề xuất dưới đây là tối thiểu về số lượng so sánh được thực hiện. Các chuỗi được giữ ở trạng thái tĩnh (trong mảng) và bản thân các mảng được tổ chức trong một danh sách tuyến tính động. Tại mỗi bước, các phần tử hiện tại của mảng được so sánh và phần tử tối thiểu được xác định. Sau đó hiển thị và tắt. Nếu một mảng trống, nó ngay lập tức bị loại khỏi danh sách các mảng và không tham gia vào các phép so sánh sau. Lần này chúng ta sẽ làm việc với các phần tử của kiểu struct CElem và chúng ta sẽ so sánh các khóa key.

Chương trình 7.18. Trộn hai mảng thứ tự làm một (718mergearr.c)

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define MAX 12
#define ARRAYS 6
struct CElem {
    int key;
```

```
/* một số dữ liệu khác*/
};

struct CList {
    unsigned len, point;
    struct CElem data[MAX];
    struct CList *next;
};

/* Điền những giá trị nguyên ngẫu nhiên */
struct CList *init(unsigned mod)
{
    struct CList *head, *p;
    unsigned i, j;
    srand(time(NULL));
    for (head = NULL, i = 0; i < ARRAYS; i++) {
        p = (struct CList *) malloc(sizeof(struct CList));
        p->len = MAX;
        p->point = 0;
        p->data[0].key = (rand() % mod);
        for (j = 1; j < MAX; j++)
            /* Tạo một chuỗi được sắp xếp*/
            p->data[j].key = p->data[j-1].key + (rand() % mod);
        p->next = head;
        head = p;
    }
    return head;
}

void merge(struct CList *head)
{
    struct CList *p, *q, *pMin;
    struct CElem k1, k2;
    int i;
    printf("\\n");
    p = (struct CList *) malloc(sizeof(struct CList));
    p->next = head;
    head = p;
    for (i = 0; i < MAX*ARRAYS; i++) {
        p = head; pMin = head;
        while (NULL != p->next) {
            k1 = p->next->data[p->next->point];
```

```

k2 = pMin->next->data[pMin->next->point];
if (k1.key < k2.key)
    pMin = p;
    p = p->next;
}
printf("%8d", pMin->next->data[pMin->next->point].key);
if (pMin->next->len-1 == pMin->next->point) {
    q = pMin->next;
    pMin->next = pMin->next->next;
    free(q);
}
else
    pMin->next->point++;
}

void print(struct CList *head)
{ unsigned i;
    for (; NULL != head; head = head->next) {
        for (i = 0; i < MAX; i++)
            printf("%6d", head->data[i].key);
        printf("\n");
    }
    printf("\n");
}

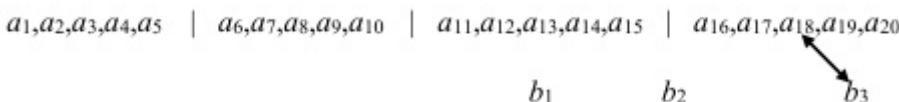
int main()
{ struct CList *head;
    head = init(500);
    printf("\nMảng trước khi sắp xếp:\n");
    print(head);
    printf("Kết quả kết hợp:");
    merge(head);
    return 0;
}

```

Rõ ràng, hàm `merge()` thực hiện $n + m$ số phép so sánh (m và n là độ dài của cả hai dãy). Tại các giá trị gần nhau của m và n , điều này là tốt. Nếu $m = n$ trong trường hợp tổng quát, chúng ta sẽ cần $2n - 1$ phép so sánh. Nhưng điều gì xảy ra ở $m = 1$? Trong trường

hợp này, sử dụng *tìm kiếm nhị phân* (xem ??) Để tìm vị trí cần chèn B_1 , chúng ta có thể giải quyết vấn đề hiệu quả hơn nhiều - phép so sánh $\log_2 n$ sẽ là đủ thay vì $\Theta(n + 1)$.

Bài toán mà chúng ta sẽ tự đặt ra là tận dụng tối đa cả hai thuật toán: cụ thể là hợp nhất đơn giản và tìm kiếm nhị phân, biến đổi một thuật toán hoạt động không tệ hơn chúng, $tim = 1$ hoạt động giống như tìm kiếm nhị phân và tại $m = n$ - như một sự hợp nhất đơn giản.



Hình 7.5. Hợp nhất nhị phân của các mảng đã sắp xếp.

Lần này C sẽ phát triển qua lại - giống như một ngăn xếp. Để xác định, giả sử $n > m$ và chia A thành $m + 1$ nhóm (dãy con) với số phần tử xấp xỉ bằng nhau. Ta so sánh B_m với phần tử cuối cùng A_k của nhóm áp chót trong A . Nếu B_m nhỏ hơn A_k , thì ta có thể chuyển A_k sang C , cùng với tất cả các phần tử ở bên phải của A_k , nghĩa là, toàn bộ nhóm cuối cùng. Nếu không ($B_m \geq A_k$) tìm vị trí chèn tương ứng của B_m bằng tìm kiếm nhị phân, sau đó chuyển sang C B_m và các phần tử của A ở bên phải B_m . Vì tìm kiếm nhị phân hoạt động tốt nhất trên các mảng có kích thước, bởi của 2, sẽ tốt hơn nếu nhóm cuối cùng chứa $2^{\lfloor \log_2 n/m \rfloor}$ phần tử thay vì n/m . Quá trình tiếp tục cho đến khi hết các phần tử của một dãy, trong đó các phần tử của dãy kia được chuyển sang C . (xem Hình 7.5)

Việc sử dụng tìm kiếm nhị phân ngũ ý chắc chắn truy cập trực tiếp vào từng phần tử của mảng và chắc chắn dẫn đến triển khai tinh, như trong hàm ứng dụng `binaryMerge()`, thực hiện hợp nhất nhị phân các mảng theo thuật toán được mô tả. Nó nhận các tham số của hai mảng đã sắp xếp `a[]` và `b[]`, cũng như số phần tử trong mỗi mảng - m và n tương ứng, và trả về một mảng `c[]`, chứa chuỗi đã hợp nhất:

Chương trình 7.19. Trộn hai mảng nhị phân (719binmerge.c)

```
int binarySearch(struct CElem m[], int left,
```

```

        int right, struct CElem elem)
{ int middle;
  do {
    middle = (left + right) / 2;
    if (m[middle].key < elem.key)
      left = middle + 1;
    else
      right = middle - 1;
  } while (left <= right);
  return right;
}

void binaryMerge(struct CElem a[], struct CElem b[],
                 struct CElem c[], int n, int m)
{ int t, t2, cind, k, j;
  cind = n + m;
  while (n > 0 && m > 0) {
    if (m <= n) {
      t = (int) (log(n / m) / log(2));
      t2 = 1 << t; /* T2<- 2^T */
      if (b[m - 1].key < a[n - t2].key) {
        /* Bỏ a[n-t2-1],...,a[n] vào dãy ban đầu */
        cind -= t2;
        n -= t2;
        for (j = 0; j < t2; j++)
          c[cind + j] = a[n + j];
      }
    }
    else {
      k = binarySearch(a, n - t2, n - 1, b[m - 1]);
      for (j = 0; j < n - k - 1; j++)
        c[cind - n + k + j + 1] = a[k + j + 1];
      cind -= n - k - 1;
      n = k + 1;
      c[--cind] = b[--m];
    }
  }
  else {
    t = (int) (log(m / n) / log(2));
    t2 = 1 << t; /* T2<- 2^T */
    if (a[n - 1].key < b[m - t2].key) {

```

```

/* Bỏ b[m-t2-1],...,b[m] vào dãy ban đầu */
cind -= t2;
m -= t2;
for (j = 0; j < t2; j++)
    c[cind + j] = b[m + j];
}
else {
    k = binarySearch(b, m - t2, m - 1, a[n - 1]);
    for (j = 0; j < m - k - 1; j++)
        c[cind - m + k + j + 1] = b[k + j + 1];
    cind -= m - k - 1;
    m = k + 1;
    c[-cind] = a[--n];
}
}
if (n == 0)
    for (j = 0; j < m; j++)
        c[j] = b[j];
else
    for (j = 0; j < n; j++)
        c[j] = a[j];
}

```

Bài tập

- 7.10. Hãy đề xuất và triển khai một thuật toán để hợp nhất hai chuỗi được sắp xếp từ một mảng tại chỗ, tức là, mà không cần sử dụng một mảng bổ sung.

7.5. Sắp xếp theo hợp nhất

Chúng ta sẽ phát triển một thuật toán để sắp xếp một mảng, và sau đó là một danh sách được liên kết dựa trên *nguyên tắc chia để trị*. Chúng ta sẽ bắt đầu với một mảng. Ý tưởng là chia nó thành hai phần, để sắp xếp chúng riêng biệt (chia), và sau đó hợp nhất chúng thành một mảng được sắp xếp chung (quy tắc). Việc hợp nhất hai phần có thể được thực hiện bằng cách chuyển qua hai mảng con đã được sắp xếp. Chúng ta sẽ thực hiện phép chia ở giữa mảng và

chúng ta sẽ áp dụng đệ quy cùng một thuật toán cho từng phần. Quá trình kết thúc khi đến một mảng con chứa 0 hoặc 1 phần tử.

Rõ ràng, hợp nhất nhị phân ở đây (xem 7.4) Không thể có ích gì cho chúng ta, vì chúng ta hợp nhất các phân vùng với một số phần tử bằng nhau. Chúng ta có một mảng duy nhất $a[]$, được chia thành hai phần mà chúng ta muốn hợp nhất. Chương trình được đề xuất sử dụng một sửa đổi của phương pháp giới hạn. (xem 7.4) Thuật toán tiến hành theo hai bước. Trong bước đầu tiên, nội dung của $a[]$ được sao chép vào mảng phụ $b[]$, và các phần tử của phần thứ hai được sao chép theo thứ tự giảm dần. Sau đó, các phép so sánh liên tiếp bắt đầu từ hai đầu đối diện của mảng. Lưu ý rằng phần tử lớn nhất sẽ nằm ở giữa mảng (nó là phần tử cuối cùng trong một trong hai phân vùng được sắp xếp) và sẽ đóng vai trò như một ràng buộc đối với cả hai chỉ mục sau khi một trong các phân vùng bị cạn kiệt. Đây là cách chúng ta nhận được chương trình sau:

Chương trình 7.20. Sắp xếp theo hợp nhất (720mergea.c)

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
int a[MAX], /* Mảng cơ sở cho sắp xếp*/
       b[MAX]; /* Mảng trợ giúp*/
const unsigned n = 100; /* Số phần tử sắp xếp*/
/* Sinh ra tập ví dụ*/
void generate(void)
{
    unsigned i;
    for (i = 0; i < n; i++)
        a[i] = rand() % (2*n + 1);
}

/* In danh sách lên màn hình*/
void printList(void)
{
    unsigned i;
    for (i = 0; i < n; i++)
        printf("%4d", a[i]);
}

/* Sắp xếp */
```

```

void mergeSort(unsigned left, unsigned right)
{ unsigned i, j, k, mid;
  if (right <= left) return; /* kiểm tra xem có gì sắp xếp*/
  mid = (right + left) / 2;
  mergeSort(left, mid); /* Sắp xếp phần bên trái */
  mergeSort(mid + 1, right); /*Sắp xếp phần bên phải*/
  /* Chép các phần tử của a[] vào mảng trợ giúp b[] */
  for (i = mid + 1; i > left; i--)
    b[i - 1] = a[i - 1]; /* Chiều bên phải */
  for (j = mid; j < right; j++)
    b[right + mid - j] = a[j + 1]; /* chiều ngược lại*/
  /* Hợp hai mảng vào a[] */
  for (k = left; k <= right; k++)
    a[k] = (b[i] < b[j]) ? b[i++] : b[j--];
}

int main()
{
  generate();
  printf("Trước khi sắp xếp:\n");
  printList();
  mergeSort(0, n-1);
  printf("Sau khi sắp xếp:\n");
  printList();
  return 0;
}

```

Nhược điểm chính của việc triển khai trên là việc sao chép dữ liệu ở mỗi bước. Điều này có thể tránh được bằng cách duy trì một số chức năng hợp nhất. Ý tưởng là xen kẽ hợp nhất a[] thành b[] với hợp nhất b[] thành a[], cũng như hợp nhất tăng dần với hợp nhất theo thứ tự giảm dần. Ví dụ, để có được một dãy được sắp xếp tăng dần trong a[], chúng ta cần hợp nhất hai dãy của b[]: dãy đầu tiên được sắp xếp theo thứ tự tăng dần (ký hiệu là b1[]) và dãy thứ hai - theo thứ tự giảm dần (b2[]). Để có được b1[], chúng ta cần hợp nhất hai chuỗi tương ứng từ phần đầu tiên của một a[] (được sắp xếp theo thứ tự tăng dần và giảm dần, tương ứng). Để có được b2[], hai phần khác của dãy a[] phải được hợp nhất (lại được sắp xếp theo thứ tự giảm dần và tăng dần, tương ứng), v.v. Cần có 4 hàm hợp nhất khác nhau cho bốn kết hợp có thể có: hợp nhất từ a[]/b[]

đến b[]/a[] theo thứ tự tăng dần/giảm dần.

Một cách khác (ít hiệu quả hơn) để tránh sao chép dữ liệu là sử dụng danh sách liên kết. Hợp nhất hai danh sách liên kết đã sắp xếp gần giống như hợp nhất hai mảng đã sắp xếp. Sau đây là một ví dụ về chức năng thực hiện việc sáp nhập.

Chương trình 7.21. Sửa sắp xếp theo hợp nhất (721mergel1.c)

```

struct list { /* Loại danh sách liên kết */
    int value;
    struct list *next;
} *empty; /* Phần tử trống */

const unsigned long n = 100;

struct list *merge(struct list *a, struct list *b)
{
    struct list *head, *tail;
    /* Giả thiết hai danh sách cửa nhau 1 phần tử */
    tail = head = empty;
    for (;;) {
        if (a->value < b->value) {
            tail ->next = a;
            a = a->next;
            tail = tail ->next;
        if (NULL == a) {
            tail ->next = b;
            break;
        }
    }
    else {
        tail ->next = b;
        b = b->next;
        tail = tail ->next;
        if (NULL == b) {
            tail ->next = a;
            break;
        }
    }
}
return head->next;
}

```

Mọi thứ khó chịu hơn một chút với việc chia danh sách thành hai danh sách con trước khi kháng cáo đệ quy. Việc thiếu quyền truy cập trực tiếp vào các phần tử của một danh sách được liên kết đòi hỏi phải có một đoạn văn rõ ràng qua một nửa các phần tử của nó. Vì mục đích này, nên chỉ định số phần tử trong phân vùng dưới dạng tham số, như được thực hiện trong hàm sau.

Sửa đổi hàm sắp xếp trong 721mergel1.c

```
struct list *mergeSort(struct list *c, unsigned long n)
{
    struct list *a, *b;
    unsigned long i, n2;
    /* Nếu danh sách chứa chỉ 1 phần tử thì không làm gì*/
    if (n < 2)
        return c;
    /* Chia danh sách làm hai phần*/
    for (a = c, n2 = n / 2, i = 2; i <= n2; i++)
        c = c->next;
    b = c->next;
    c->next = NULL;
    /* Sắp xếp riêng hai phần sau đó hợp*/
    return merge(mergeSort(a, n2), mergeSort(b, n - n2));
}
```

Sử dụng phương pháp giới hạn cho phép tối ưu hóa quá trình sáp nhập. Vì mục đích này, chúng ta sẽ bỏ chỉ báo tiêu chuẩn cuối danh sách NULL. Thay vào đó, chúng ta sẽ sử dụng một phần tử đặc biệt z, mà con trỏ tiếp theo sẽ trỏ đến z và value của nó sẽ là giá trị tối đa cho phép đối với kiểu (trong trường hợp này là INFINITY). Tất cả các danh sách sẽ có z là phần tử cuối cùng của chúng.

Chúng ta đã đề cập ở trên rằng hàm mergeSort() cần nhận như một tham số là số lượng các mục trong danh sách. Trong trường hợp chung, nó có thể không được biết trước. Một giải pháp khả thi sau đó là tìm trước số bằng cách duyệt qua danh sách và sau đó gửi nó dưới dạng tham số, như trên. Một cách tiếp cận khả thi khác là tổ chức một chu trình bao gồm hai con trỏ: một con di chuyển ở bước 1 và con thứ hai ở bước 2. Khi con trỏ thứ hai đến cuối danh sách, con trỏ đầu tiên sẽ trỏ vào giữa. Lưu ý rằng nếu được sử dụng làm dấu phân tách NULL, thuật toán này sẽ không hoạt động bình thường,

như trong trường hợp số lượng mục chẵn, phần cuối của danh sách sẽ bị bỏ qua. Tuy nhiên, trong trường hợp của chúng ta, trường tiếp theo của z lại trả tới z, vì vậy không có gì nguy hiểm. Do đó, chúng ta nhận được phiên bản sau của chương trình:

Chương trình 7.22. Sửa sắp xếp theo hợp nhất (722mergel2.c)

```
#include <stdio.h>
#include <stdlib.h>
#define INFINITY (int)((1 << (sizeof(int)*8 - 1)) - 1)
const unsigned long n = 100;
struct list { /*Loại danh sách liên kết */
    int value;
    struct list *next;
} *z; /* pàn tử trống */

/* Tạo ra tập ví dụ */
struct list *generate(unsigned long n)
{ struct list *p, *q;
unsigned long i;
for (p = z, i = 0; i < n; i++) {
    q = (struct list *) malloc(sizeof(struct list));
    q->value = rand() % (2*n + 1);
    q->next = p;
    p = q;
}
return p;
}

void printList(struct list *p) /*In danh sách lên màn hình */
{ for (; p != z; p = p->next)
    printf("%4d", p->value);
}

struct list *merge(struct list *a, struct list *b)
{ struct list *c;
c = z;

/* Giả thiết hai danh sách chứa ít nhất 1 phần tử */
do {
    if (a->value < b->value) {
        c->value = a->value;
        a = a->next;
    } else {
        c->value = b->value;
        b = b->next;
    }
    c->next = merge(a, b);
} while (a != z || b != z);
return c;
}
```

```

    c->next = a;
    c = a;
    a = a->next;
}
else {
    c->next = b;
    c = b;
    b = b->next;
}
} while (c != z);
c = z->next;
z->next = z;
return c;
}

struct list *mergeSort(struct list *c)
{
struct list *a, *b;
/* Nếu danh sách chứa chỉ 1 phần tử: không làm gì cả */
if (c->next == z)
    return c;
/* Danh sách chia làm hai phần */
for (a = c, b = c->next->next->next; b != z; c = c->next)
    b = b->next->next;
    b = c->next;
    c->next = z;
/* Sắp xếp riêng từng phần sau đó hợp */
return merge(mergeSort(a), mergeSort(b));
}

int main()
{
struct list *l;
/* Khởi tạo z */
z = (struct list *) malloc(sizeof(struct list));
z->value = INFINITY;
z->next = z;

l = generate(n);
printf("Trước khi sắp xếp:\n");
printList(l);
l = mergeSort(l); /* giả thiết danh sách chứa ít nhất 1 phần tử */
}

```

```

printf("Sau khi sắp xếp:\n");
printList(1);
return 0;
}

```

Ai cũng biết rằng mọi chương trình đệ quy đều có một phép lặp tương đương. Đối với một số chương trình, việc chuyển đổi từ đệ quy sang lặp lại rất dễ dàng và đơn giản. Đối với những người khác, đây có thể là một bài toán khá khó khăn. Một cách tiêu chuẩn để chuyển đổi một chương trình đệ quy thành một chương trình lặp là sử dụng một ngăn xếp. May mắn thay, `mergeSort()` cho phép thực hiện lặp đi lặp lại tương đối đơn giản mà không cần ngăn xếp.

Phương thức thực hiện lặp được đề xuất không hoàn toàn tương đương với phương pháp đệ quy và thực hiện các phép chia theo những cách khác nhau. Quá trình này được điều khiển bởi vòng lặp `for` bên ngoài. Trong bước đầu tiên, các cặp phần tử có thứ tự được hình thành, trong - phần thứ hai, trong - phần thứ ba, v.v. Ở mỗi bước tiếp theo, các cặp danh sách có thứ tự liên tiếp được hợp nhất, dẫn đến các danh sách được sắp xếp mới có độ dài gấp đôi so với bước trước đó. Chúng ta sẽ giữ tất cả các danh sách phụ được liên kết trong một danh sách chung. Việc sát nhập diễn ra theo chu trình nội bộ. Một danh sách việc `todo` gồm các phần tử chưa xử lý được hình thành. Vòng lặp `while` nội bộ tuần tự tìm kiếm các cặp danh sách thu được ở bước trước, sau đó hợp nhất chúng. Danh sách kết quả được thêm vào cuối danh sách các phần tử đã xử lý. Việc thực hiện chu trình nội bộ tiếp tục cho đến khi tất cả các mục trong danh sách này được chuyển giao. [Sedgewick-1992]

Chương trình 7.23. Sửa sắp xếp theo hợp nhất (723mergel3.c)

```

struct list *mergeSort(struct list *c)
{ unsigned long i, n, n2;
  struct list *a, *b, *head, *todo, *t;
  head = (struct list *) malloc(sizeof(struct list));
  head->next = c;
  a = z;
  for (n = 1; a != head->next; n <= 1) {
    todo = head->next;
    c = head;

```

```

while (todo != z) {
    t = todo;
    /* chia mảng a[] */
    for (a = t, i = 1; i < n; i++)
        t = t->next;
    /* chia mảng b[] */
    b = t->next; t->next = z;
    for (t = b, i = 1; i < n; i++)
        t = t->next;
    /* hợp a[] và b[] */
    todo = t->next; t->next = z;
    c->next = merge(a, b);
    /* Bỏ qua mảng đã hợp nhất*/
    for (n2 = n + n, i = 1; i <= n2; i++)
        c = c->next;
}
}
return head->next;
}

```

Hiệu quả của sắp xếp hợp nhất là gì? Nó chỉ ra rằng đây là một trong những thuật toán sắp xếp hiệu quả nhất đã biết với độ phức tạp thời gian trung bình được đảm bảo $\Theta(n \log_2 n)$. Về mặt lý thuyết, đây là độ phức tạp thuật toán tốt nhất có thể đạt được bằng một thuật toán sắp xếp phổ quát (xem ??). Như chúng ta đã thấy trong đoạn ??, Với các giả định bổ sung cho biểu diễn dữ liệu (nhị phân), các thuật toán sắp xếp tuyến tính có thể thu được. Tuy nhiên, chúng không phổ biến và không thể được sử dụng để sắp xếp các số dấu phẩy động, chẳng hạn. Điều này đặt nó bên cạnh hình chóp (xem ??) và nhanh (xem ??), có cùng độ phức tạp thời gian trung bình. Hơn nữa, tương tự như sắp xếp theo hình chóp, sắp xếp hợp nhất có cùng độ phức tạp trong trường hợp xấu nhất (với hằng số ẩn nhỏ hơn hằng số hình chóp).

Đồng thời, sắp xếp nhanh trong trường hợp xấu nhất có độ phức tạp $\Theta(n^2)$, xếp nó trong số các thuật toán kém hiệu quả nhất như phương pháp bong bóng, xem ?? (Cái sau, như đã đề cập, là cơ sở của *sắp xếp nhanh*.) Khi chúng ta muốn đảm bảo độ phức tạp $\theta(n \log_2 n)$, kể cả trong trường hợp xấu nhất, chúng ta có thể sử dụng *sắp xếp*

theo hình chót hoặc sắp xếp hợp nhất.

Một nhược điểm chính của sắp xếp hợp nhất là nó yêu cầu bộ nhớ bổ sung tỷ lệ với n : cho một mảng bổ sung hoặc cho các con trỏ, tương ứng. Các thử nghiệm thực nghiệm cho thấy rằng, trong trường hợp trung bình, phân loại nhanh đập hình chót từ 2-3 lần (xem [Wirth-1980]). Tốc độ sắp xếp hợp nhất chiếm một vị trí trung gian giữa hai, đó là lý do tại sao nó là một ứng cử viên nặng ký khi tốc độ là quan trọng trong trường hợp xấu nhất, đồng thời có khả năng cấp phát bộ nhớ bổ sung tỷ lệ với n .

Một ưu điểm chính khác của sắp xếp hợp nhất là quyền truy cập vào các phần tử được thực hiện *tuần tự nghiêm ngặt*, đó là lý do tại sao nó phù hợp để sắp xếp danh sách, tệp liên tiếp, v.v. Cuối cùng nhưng không kém phần quan trọng, cần lưu ý rằng việc sắp xếp theo cách hợp nhất là ổn định, tức là nó bảo toàn thứ tự tương đối của các phần tử có khóa bằng nhau. Đây là một thuộc tính quan trọng và không dành riêng cho mọi thuật toán sắp xếp. Ví dụ, phân loại nhanh là không ổn định và đòi hỏi nhiều nỗ lực để làm cho nó ổn định. Một tính năng thú vị khác của sắp xếp bằng cách hợp nhất trong biến thể của nó, khi việc hợp nhất không kiểm tra xem một trong hai danh sách có bị cạn kiệt hay không, là độ phức tạp của nó không phụ thuộc vào sự sắp xếp sơ bộ của các phần tử của tập hợp. Trong phiên bản có sự kiểm tra, sắc lệnh có ảnh hưởng không đáng kể.

Giống như hầu hết các phương pháp sắp xếp hiện đại, sắp xếp hợp nhất hoạt động kém với một số phần tử nhỏ. Cũng như phân loại nhanh, kết quả tốt sẽ thu được khi kết hợp với các phương pháp khác. Một cách tiếp cận tiêu chuẩn là sử dụng sắp xếp chèn khi số phần tử của tập hợp được đề cập giảm xuống dưới 15-20 (con số này là một ví dụ) (xem ??).

Bài tập

▷ 7.11. So sánh tốc độ triển khai được đề xuất để sắp xếp bằng cách hợp nhất cho: 100; 1000; 10000; 100.000 phần tử.

▷ 7.12. Viết chương trình sắp xếp theo hợp nhất, trong đó tập hợp được chia thành:

- a) 3 tập hợp con
- b) k tập hợp con, $k > 3$

▷ **7.13.** Tính toán mức độ phức tạp của các phương án sáp nhập khác nhau:

- (a) tốt nhất
- b) trong trường hợp xấu nhất

Tìm chuỗi đầu vào tốt nhất và kém nhất.

▷ **7.14.** Hãy triển khai một biến thể lặp đi lặp lại của sáp xếp bằng cách hợp nhất, hãy sử dụng mảng thay vì danh sách được liên kết.

▷ **7.15.** Hãy xác định bằng thực nghiệm số lượng phần tử mà sáp xếp chèn nên được sử dụng để sáp xếp hợp nhất.

▷ **7.16.** Hãy so sánh thực nghiệm tốc độ sáp xếp bằng cách hợp nhất, kết hợp với:

- (a) sáp xếp chèn
- b) phương pháp bong bóng
- c) lựa chọn trực tiếp

▷ **7.17.** Hãy thực hiện một biến thể sáp xếp bằng cách hợp nhất, lợi dụng của sự sáp xếp tự nhiên có thể có của các phần tử trong tập nguồn (xem [7.4](#)).

7.6. Nâng nhanh lũy thừa

Trong 1.1.1. chúng ta đã xem xét một cách lặp lại đơn giản để tìm x^n (x - số thực, n - tự nhiên). Có một thuật toán nhanh hơn (thực hiện chia tỷ lệ với ít phép nhân hơn) dựa trên công thức sau [Schwertner-1995]:

$$x^n = \begin{cases} x^{\frac{n}{2}} \cdot x^{\frac{n}{2}}, & \text{với } n \text{ là số chẵn} \\ x^{n-1} \cdot x, & \text{với } n \text{ là số lẻ} \end{cases}$$

Không khó để thấy rằng công thức trên dựa trên *phép chia để trị*, tuy ở dạng hơi lạ: ở mỗi bước ta có hai bài toán con, nhưng thực tế chỉ cần giải một bài không tầm thường (đối với *phân chia*). Trong

trường hợp đầu tiên, vì hai bài toán con trùng nhau, và trong trường hợp thứ hai - vì một bài toán nhỏ. Tuy nhiên, phần *để trị* là cổ điển và dựa trên các giải pháp của hai bài toán con, một giải pháp cho vấn đề ban đầu sẽ thu được.

Chương trình 7.24. Nâng nhanh lũy thừa (724powerc.c)

```
#include <stdio.h>
const double base = 3.14;
const unsigned d = 11;
double power(double x, unsigned n)
{ if (0 == n) return 1;
  else
    if (n % 2)
      return x * power(x, n - 1);
    else
      return power(x * x, n / 2);
}

int main()
{
  printf("%lf^%u = %lf\n", base, d, power(base, d));
  return 0;
}
```

Chúng ta sẽ lưu ý rằng công thức trên không cung cấp số nhân tối thiểu để thực hiện nâng theo độ. Ví dụ, đối với $n = 15$, các chương trình trên sẽ thực hiện việc nâng tuần tự như sau:

- 1) x^1
- 2) $x^2 = x^1 \cdot x^1$
- 3) $x^3 = x^2 \cdot x^1$
- 4) $x^6 = x^3 \cdot x^3$
- 5) $x^7 = x^6 \cdot x^1$
- 6) $x^{14} = x^7 \cdot x^7$
- 7) $x^{15} = x^{14} \cdot x^1$

tức là với tổng số là 7 phép nhân. Có các giải pháp khác, ví dụ:

- 1) x^1
- 2) $x^2 = x^1 \cdot x^1$
- 3) $x^3 = x^2 \cdot x^1$
- 4) $x^4 = x^3 \cdot x^1$
- 5) $x^7 = x^4 \cdot x^3$

$$6) x^8 = x^7 \cdot x^1$$

$$7) x^{15} = x^8 \cdot x^7$$

Tuy nhiên, cũng có những chuỗi ngắn hơn chỉ gồm 6 phép nhân, ví dụ:

$$1) x^1$$

$$2) x^2 = x^1 \cdot x^1$$

$$3) x^3 = x^2 \cdot x^1$$

$$4) x^5 = x^3 \cdot x^2$$

$$5) x^{10} = x^5 \cdot x^5$$

$$6) x^{15} = x^{10} \cdot x^5$$

Như :

$$1) x^1;$$

$$2) x^2 = x^1 \cdot x^1;$$

$$3) x^4 = x^2 \cdot x^2;$$

$$4) x^5 = x^4 \cdot x^1;$$

$$5) x^{10} = x^5 \cdot x^5;$$

$$6) x^{15} = x^{10} \cdot x^5$$

Nếu chúng ta chỉ xem xét một loạt các chỉ số theo cấp số nhân:

1, 2, 3, 4, 7, 8, 15

1, 2, 3, 5, 10, 15

1, 2, 4, 5, 10, 15

có thể thấy rằng đây là những hàng trong đó phần tử đầu tiên là 1, và mỗi phần tiếp theo là tổng của hai phần trước nó. Một loạt như vậy được gọi là phụ gia. Thật không may, cách duy nhất đã biết để tìm một chuỗi cộng với độ dài tối thiểu là cạn kiệt hoàn toàn, điều này không được quan tâm trong đoạn này. [Nakov-1998]

Bài tập

▷ 7.18. Đề xuất một thuật toán heuristic (xem Chương 9) luôn tìm thấy một dãy cộng không dài hơn dãy được đưa ra bởi công thức trên.

7.7. Thuật toán Strassen để nhân nhanh các ma trận

Phép nhân các ma trận là một cơ hội mới để chứng minh các khả năng của nguyên lý La Mã cũ. Chúng ta sẽ nhớ lại rằng thuật toán cổ điển để nhân ma trận $A = (a_{ij})_{m \times n}$ và $B = (b_{ij})_{n \times r}$ bắt đầu trực tiếp từ định nghĩa và được đưa ra bởi công thức:

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}.$$

Trong trường hợp ma trận vuông có kích thước $n \times n$, thời gian tính toán của ma trận C hiển nhiên là $\Theta(n^3)$: ta có n^2 phần tử và việc tính toán mỗi phần tử cần thời gian $\Theta(n)$. Tất nhiên, ở đây, chúng ta giả sử rằng phép cộng và phép nhân có độ phức tạp theo thời gian không đổi $\Theta(1)$, không phụ thuộc vào n .

Thuật toán này đơn giản và hiển nhiên đến nỗi trong một thời gian dài không ai nghĩ rằng có thể có một thuật toán khác và không cố gắng tìm kiếm nó. Tuy nhiên, vào cuối những năm 1960, Strassen đã đề xuất một cải tiến thú vị cho thuật toán, giảm độ phức tạp xuống $\log(n \log^7)$. Khám phá của ông gây ngạc nhiên cho giới khoa học và dẫn đến *phương pháp chia để trị* rất xứng đáng, vốn cho đến nay vẫn bị đánh giá thấp như một kỹ thuật lập trình hiệu quả. Hiệu quả to lớn của khám phá Strassen là do phép nhân các ma trận, như một phép toán cơ bản của đại số, được đưa vào nhiều thuật toán số. Bất kỳ cải tiến nào ở đây sẽ tự động làm giảm độ phức tạp tính toán của một số thuật toán khác.

Xét các ma trận A và B có kích thước 2×2 :

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \times \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$

Sử dụng thuật toán tiêu chuẩn, chúng ta thu được:

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j}, i = 1, 2 \text{ và } j = 1, 2$$

Có thể thấy rằng chúng ta cần 8 phép nhân và 4 phép cộng. Strassen lưu ý rằng số phép nhân có thể giảm xuống còn 7 như

sau (với P_i , chúng ta có nghĩa là các biến phụ):

$$\begin{aligned}
 P_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
 P_2 &= (a_{21} + a_{22})b_{11} \\
 P_3 &= a_{11}(b_{12} - b_{22}) \\
 P_4 &= a_{22}(b_{21} - b_{11}) \\
 P_5 &= (a_{11} + a_{12})b_{22} \\
 P_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\
 P_7 &= (a_{12} - a_{22})(b_{21} + b_{22}) \\
 c_{11} &= P_1 + P_4 - P_5 + P_7 \\
 c_{12} &= P_3 + P_5 \\
 c_{21} &= P_2 + P_4 \\
 c_{22} &= P_1 + P_3 - P_2 + P_6
 \end{aligned}$$

Tất nhiên, không có bữa trưa miễn phí: số lượng phép cộng (bao gồm cả phép trừ) tăng lên 18. Điều này đặt ra câu hỏi về lợi thế của phương pháp Strassen trong trường hợp này, do thực tế là các bộ xử lý cũ hơn "nhân" nhiều như hai (hiếm khi là ba.) phép cộng / phép trừ. Tuy nhiên, hầu hết các bộ vi xử lý hiện đại đều thực hiện phép cộng và phép trừ ở cùng một tốc độ. Đó là, phương pháp được trình bày dưới đây không dẫn đến cải tiến, ít nhất là không trực tiếp. Tuy nhiên, Strassen lưu ý rằng thuật toán có thể được áp dụng một cách đệ quy. Trước khi thảo luận về cách điều này có thể xảy ra, chúng ta sẽ chỉ ra rằng chuỗi hoạt động trên không phải là duy nhất. Một khả năng hiện thực hóa khác của ý tưởng trông như thế

này (xem [Brassard, Bratley-1987]):

$$\begin{aligned}
 P_1 &= (a_{21} + a_{22} - a_{11})(b_{22} - b_{12} + b_{11}) \\
 P_2 &= a_{11}b_{11} \\
 P_3 &= a_{12}b_{21} \\
 P_4 &= (a_{11} - a_{21})(b_{22} - b_{12}) \\
 P_5 &= (a_{21} + a_{22})(b_{12} - b_{11}) \\
 P_6 &= (a_{12} - a_{21} + a_{11} - a_{22})b_{22} \\
 P_7 &= a_{22}(b_{11} + b_{22} - b_{12} - b_{21}) \\
 c_{11} &= P_2 + P_3 \\
 c_{12} &= P_1 + P_2 + P_5 + P_6 \\
 c_{21} &= P_1 + P_2 + P_4 - P_7 \\
 c_{22} &= P_1 + P_2 + P_4 + P_5
 \end{aligned}$$

Tất nhiên, có những cách khác để đạt được hiệu quả tương tự. Chúng ta sẽ không xem xét tất cả chúng, nhưng chúng ta vẫn sẽ bị cám dỗ để trích dẫn một cái khác ([Aho, Hopcroft, Ullman-1987]):

$$\begin{aligned}
 P_1 &= (a_{12} - a_{22})(b_{21} + b_{22}) \\
 P_2 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
 P_3 &= (a_{11} - a_{21})(b_{11} + b_{12}) \\
 P_4 &= (a_{11} + a_{12})b_{22} \\
 P_5 &= a_{11}(b_{12} - b_{22}) \\
 P_6 &= a_{22}(b_{21} - b_{11}) \\
 P_7 &= (a_{21} + a_{22})b_{11} \\
 c_{11} &= P_1 + P_2 - P_4 + P_6 \\
 c_{12} &= P_4 + P_5 \\
 c_{21} &= P_6 + P_7 \\
 c_{22} &= P_2 - P_3 + P_5 - P_7
 \end{aligned}$$

Nếu chúng ta thay thế các phần tử a_{ij} , b_{ij} và c_{ij} bằng các ma trận có kích thước $n \times n$, chúng ta nhận được một thuật toán có thể nhân các ma trận có kích thước $2n$, sử dụng 7 thay vì 8 phép nhân của các ma trận có kích thước $n \times n$. Tuy nhiên, nhân hai ma trận chắc

chắn là một phép toán khó hơn phép cộng. Thật vậy, nếu sử dụng các thuật toán cổ điển từ các định nghĩa của phép cộng, chúng ta cần n^2 tổng cơ bản, thì đối với phép nhân chúng ta cần n^3 tổng cơ bản cộng với n^3 phép nhân sơ cấp. Đó là, lưu một phép nhân có ảnh hưởng nghiêm trọng.

Nhưng tại sao lại dừng ở đó? Nếu n là một số chẵn, chúng ta có thể áp dụng cùng một sơ đồ đệ quy cho các ma trận có kích thước $(n/2) \times (n/2)$. Trong trường hợp n là lũy thừa của 2, quá trình có thể tiếp tục cho đến khi đạt được ma trận có kích thước 2×2 . Do đó, chúng ta nhận được một thuật toán đệ quy, mà cây ở mỗi bước có phân nhánh là 7 và được duyệt theo chiều sâu. Độ phức tạp $T(n)$ của thuật toán được mô tả trong trường hợp ma trận vuông cỡ n , bậc 2, được cho bởi sự phụ thuộc lặp lại:

$$T(n) = 7T(n/2) + 18(n/2)^2, n > 2$$

Do đó theo định lý cơ bản (xem 1.4.10.) Ta thu được $T(n) \in \Theta(n^{\log 7})$. Nhưng $\log 7 \approx 2,81 < 3$, tức là Thuật toán Strassen có độ phức tạp về thời gian tốt hơn so với thuật toán cổ điển.

Điều gì sẽ xảy ra nếu chúng ta có ma trận vuông nhưng n không phải là lũy thừa của 2? Một giải pháp tiêu chuẩn cho vấn đề này là bổ sung các ma trận không có hàng và cột vào ít nhất n , đó là lũy thừa của 2. Kỹ thuật này được gọi là phép cộng tinh. Điều này có thể dẫn đến việc tăng tối đa kích thước của ma trận lên gấp đôi. Và thực hiện một phép lặp khác, theo sự phụ thuộc lặp lại ở trên, có thể tăng số lần nhân lên tối đa 7 lần.

Kể từ khi thuật toán Strassen được công bố, một số nỗ lực thành công và không thành công đã được thực hiện để cải thiện nó. Điểm yếu của thuật toán nằm ở đâu và chúng có thể được cung cấp như thế nào? Một vấn đề đã trở nên rõ ràng: việc bổ sung trước ma trận không hiệu quả, có thể làm chậm nó tới 7 lần. Vấn đề thứ hai cũng đã được giải quyết ngay từ đầu: tại $n = 2$, chúng ta thực tế có sự suy giảm thay vì cải thiện. Và $n = 3, 4, \dots$ thì sao? Ở đâu (với n) chúng ta nên ngừng áp dụng đệ quy thuật toán Strassen và sử dụng thuật toán cổ điển ở đâu?

Một giải pháp khả thi cho vấn đề đầu tiên (n không phải là lũy thừa của 2) là hoãn việc thêm ma trận với số không cho đến khi điều

này có thể thực hiện được. Tức là, nếu ban đầu n là chẵn, chúng ta có thể thực hiện một phép lặp của thuật toán. Nếu $n/2$ chẵn, chúng ta có thể làm một cái khác, và nếu nó là lẻ - chúng ta sẽ phải bổ sung nếu cần. Tuy nhiên, chỉ bằng cách bổ sung ngay bây giờ, chúng ta cuối cùng đã tiết kiệm được việc chèn thêm nhiều số không. Kỹ thuật này được gọi là bổ sung năng động.

Mặc dù dẫn đến cải thiện đáng kể, nhưng bổ sung động không phải là phương pháp tối ưu. Huss và Lederman sử dụng một kỹ thuật khác để xử lý các kích thước kỳ lạ: bóc tách động. Ý tưởng là tách một hàng và một cột. Ma trận "bóc tách" thu được có kích thước chẵn n và bước đệm quy của Strassen có thể được áp dụng cho nó. (xem Hình 7.6)

$$\left[\begin{array}{c|c} A_{11} & a_{12} \\ \hline a_{21} & a_{22} \end{array} \right] \left[\begin{array}{c|c} B_{11} & b_{12} \\ \hline b_{21} & b_{22} \end{array} \right]$$

Hình 7.6. Động "bóc tách" của A và B .

Kết quả thu được sau đó phải được kết hợp với ma trận kết quả C_{11} (xem Hình 7.7).

$$\left[\begin{array}{c|c} C_{11} & c_{12} \\ \hline c_{21} & c_{22} \end{array} \right] = \left[\begin{array}{c|c} A_{11}B_{11} + a_{12}b_{21} & A_{11}b_{12} + a_{21}b_{22} \\ \hline a_{21}B_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{array} \right]$$

Hình 7.7. Kết quả của tích của A và B .

Vấn đề thứ hai không thể được tấn công trực tiếp, vì nó yêu cầu so sánh giữa tốc độ của thuật toán cổ điển và tốc độ của thuật toán Strassen tại một n cố định. Điều này đòi hỏi phải tính đến thời gian truy cập vào các phần tử ma trận trong cả hai phương pháp, điều này phụ thuộc vào máy. Một nghiên cứu năm 1996 của Luoma

và Spider cho thấy số lượng bộ nhớ đệm có tầm quan trọng lớn và trong hầu hết các trường hợp thực tế, thuật toán ban đầu của Strassen hoạt động kém hơn thuật toán cổ điển. Họ đưa ra ý tưởng về việc triển khai thuật toán, trong đó các phép tính trung gian được thực hiện ngay trước khi chúng cần thiết, điều này đảm bảo rằng chúng vẫn nằm trong bộ nhớ cache cho đến lần sử dụng tiếp theo.

Một cải tiến bất ngờ khác của thuật toán là Vinograd. Ông đã cố gắng giảm số lượng các phép cộng / trừ từ 18 xuống còn 15, giữ nguyên số phép nhân. Đề án của anh ta có dạng:

$$\begin{aligned}
 S_1 &= a_{21} + a_{22} \\
 S_2 &= S_1 \cdot a_{11} \\
 S_3 &= a_{11} \cdot a_{21} \\
 S_4 &= a_{12} \cdot S_2 \\
 T_1 &= b_{12} \cdot b_{11} \\
 T_2 &= b_{22} \cdot T_1 \\
 T_3 &= b_{22} \cdot b_{12} \\
 T_4 &= b_{21} \cdot T_2 \\
 P_1 &= a_{11} b_{11} \\
 P_2 &= a_{12} b_{21} \\
 P_3 &= S_1 T_1 \\
 P_4 &= S_2 T_2 \\
 P_5 &= S_3 T_3 \\
 P_6 &= T_4 b_{22} \\
 P_7 &= a_{22} T_4 \\
 U_1 &= P_1 + P_4 \\
 U_2 &= U_1 + P_5 \\
 U_3 &= U_1 + P_3 \\
 c_{11} &= P_1 + P_2 \\
 c_{21} &= U_2 + P_7 \\
 c_{22} &= U_2 + P_3 \\
 c_{12} &= U_3 + P_6
 \end{aligned}$$

Một số nỗ lực khác đã được thực hiện để giảm số lượng phép nhân

và phép cộng trong sơ đồ cổ điển của Strassen ở $n = 2$, nhưng tất cả đều không thành công. Năm 1971, Hopcroft và Kerr đã chứng minh rằng điều này là không thể nếu không sử dụng phép nhân giao hoán (phép nhân ma trận không giao hoán). Sau đó, người ta đã tìm ra một cách để nhân ma trận 3×3 với tối đa 21 phép nhân, thuật toán này đưa ra một thuật toán có độ phức tạp $\Theta(n^{\log_3 21})$, tức là $\Theta(n^{2.771244})$. Có một số cải tiến khác với việc tăng kích thước ma trận và ít cơ hội áp dụng thực tế hơn. Ví dụ, Pan đã tìm ra cách nhân 70×70 ma trận với 143.640 phép nhân sơ cấp (so với 343.000, theo thuật toán cổ điển). Sau đó, các thuật toán thậm chí còn hiệu quả hơn về mặt lý thuyết và thậm chí không thể áp dụng được trong thực tế: $\Theta(n^{2.521813})$ - 1979, $\Theta(n^{2.521801})$ - 1980, $\Theta(n^{2.376})$ - 1986

Bài tập

- ▷ 7.19. Thực nghiệm để xác định thời điểm gián đoạn đệ quy và áp dụng thuật toán cổ điển.
- ▷ 7.20. Hãy kiểm tra trình tự do Vinograd để xuất để nhân nhanh các ma trận.
- ▷ 7.21. Hãy đề xuất một trình tự khác để nhân nhanh ma trận.

7.8. Nhân nhanh các số dài

Thuật toán cổ điển để nhân các số được học ở trường yêu cầu thời gian của bậc $\Theta(n^2)$ để nhân hai số có n chữ số (chúng ta giả định rằng phép cộng và nhân các số có một chữ số diễn ra trong một thời gian không đổi độc lập với n). Chúng ta đã quá quen với thuật toán này đến nỗi hầu như không ai trong chúng ta nghĩ đến tính hiệu quả và khả năng cải tiến của nó.

Hãy xem *phương pháp chia để trị* có thể mang lại cho chúng ta điều gì. Giả sử chúng ta muốn nhân hai số có n chữ số X và Y , và bây giờ chúng ta coi n là lũy thừa của 2 (chúng ta sẽ hướng dẫn bạn cách thực hiện điều này sau khi trường hợp này không xảy ra). Chia X và Y thành hai phần A, B và C, D , mỗi phần có $n/2$ chữ số sao

cho:

$$X = 2^{n/2} \cdot A + B$$

$$Y = 2^{n/2} \cdot C + D$$

Từ đây cho tích XY , chúng ta nhận được:

$$XY = 2^n \cdot AC + 2^{n/2} \cdot (AD + BC) + BD$$

Như vậy, để tính giá trị của XY theo công thức trên, cần 4 phép nhân các số có $n/2$ chữ số, 3 phép cộng các số có tối đa $n+1$ chữ số và hai lần dịch sang trái ở $n/2$ vị trí (phép nhân với 2^n và $2^{n/2}$). Nếu chúng ta sử dụng đệ quy cùng một công thức để tính các tích AC, AD, BC và BD , và quá trình tiếp tục cho đến khi chúng ta đạt đến trường hợp cơ bản của hai số có một chữ số, chúng ta nhận được sự phụ thuộc lặp lại:

$$T(1) = 1$$

$$T(n) = 4T(n/2) + cn, n > 1 \text{ và } n - \text{bậc } 2$$

Áp dụng định lý cơ bản (xem 1.4.10.), Chúng ta thu được $T(n) \in \Theta(n^2)$, tức là độ phức tạp trùng với độ phức tạp của thuật toán ban đầu, có nghĩa là phép chia đơn giản không dẫn đến cải tiến: chúng ta cũng cần một chút tháo vát. Chúng ta hãy nhớ lại thuật toán Strassen ở trên (xem ??), Trong đó sự cải tiến đến từ việc giảm số lượng phép nhân. Hãy xem liệu chúng ta có thể áp dụng nguyên tắc tương tự ở đây không. Nếu chúng ta xem xét kỹ hơn công thức, chúng ta sẽ thấy rằng chúng ta không nhất thiết phải tính các tích AD và BC : tổng của chúng là đủ. Hãy xem xét công thức:

$$XY = 2^n \cdot AC + 2^{n/2} \cdot [(A - B)(D - C) + AC + BD] + BD$$

Chúng ta sẽ không triển khai thuật toán hoạt động, vì điều này liên quan đến các hoạt động với "số dài" (tức là không phù hợp với các kiểu tiêu chuẩn), điều này sẽ làm lộn xộn mã. (Hơn nữa, một cách triển khai tốt có thể sẽ không sử dụng hệ thống số thập phân mà là hệ thống số tối đa dựa trên lũy thừa của 2, cho phép kết hợp kết quả của phép nhân hai chữ số thành một từ máy. Tuy nhiên, điều

này nằm ngoài phạm vi của sự cân nhắc của chúng ta.) chúng ta sẽ cố gắng làm rõ điều đó bằng một mã giả. Chúng ta giả sử rằng X và Y là các số nguyên có dấu và n là lũy thừa của 2. Trong trường hợp giới hạn $n = 1$, chúng ta sẽ nhận các số trực tiếp. [Aho, Hopcroft, Ullman - 1987]

Giả mã nhân số dài

```

int mult (int X, Y, n)
{
    int s; /* dấu hiệu của tích XY */
    int m1, m2, m3; /* ba tích */
    int A, B, C, D; /* một nửa của X và Y */
    /* kiểm tra trường hợp biên */
    if (1 == n)
        return (X * Y);
    s = sign(X) * sign(Y);
    X = abs (X);
    Y = abs (Y);
    A = n/2 bit bên trái của X;
    B = n/2 bit phải của X;
    C = n/2 bit bên trái của Y;
    D = n/2 bit phải của Y;
    m1 = mult (A, C, n/2);
    m2 = mult (A-B, D-C, n/2); /* *** */
    m3 = mult (B, D, n/2);
    return s * (m1 << n + (m1 + m2 + m3) << (n / 2) + m3); /* *** */
}

```

Chúng ta cũng có thể sử dụng phiên bản thứ hai của công thức, trong đó các dòng được đánh dấu bằng /* *** */ sẽ được thay thế bằng các dòng sau:

```

m2 = mult(A+B,C+D,n/2); /* *** */
return s * (m1<<n + (m2-m1-m3)<<(n/2) + m3); /* *** */

```

Điều gì xảy ra khi n không phải là lũy thừa của 2? Nếu n chẵn thì không có vấn đề gì ở bước đầu tiên và chúng ta có thể chia số thành hai phần có độ dài bằng nhau. Nếu $n/2$ chẵn, chúng ta có thể tiếp tục như vậy, nhưng nó như thế ở một bước nào đó chúng ta sẽ nhận được độ dài lẻ và điều này sẽ không thể thực hiện được. Một giải pháp rõ ràng cho vấn đề là bổ sung số bên trái với số 0 ở đầu.

Mặc dù điều này làm giảm hiệu quả của thuật toán, nhưng phân tích cho thấy rằng độ phức tạp của nó vẫn là $\Theta(n^{\log_2 3})$.

Một điểm thú vị khác đã bị bỏ qua trong phân tích ở trên là thực tế là tổng của hai số có $n/2$ chữ số không nhất thiết cho một số có n chữ số, nhưng có thể cho $(n+1)$ chữ số. Khi đó, tích $(A+B)(C+D)$, trong phiên bản thứ hai của công thức, có thể là $(n+1)$ - ở dạng số (các tích khác AC và BD không có cơ hội như vậy.). Lấy $X = 9999$ và $Y = 9998$. Ta có:

$$A = 99, B = 99, C = 99 \text{ và } D = 98$$

$$AC = 99.99 = 9801$$

$$BD = 99.98 = 9702$$

$$(A+B)(C+D) = 198.197 = 39006 \text{ (nghĩa là 5-số chữ số)}$$

May mắn thay, mức tăng nhiều nhất là 1 và độ phức tạp cuối cùng của thuật toán và trong trường hợp này sẽ là $\Theta(n^{\log_2 3})$. Chúng ta sẽ không phân tích chi tiết về những trường hợp này. Độc giả tò mò có thể tìm thêm chi tiết trong [Brassard, Bratley - 1987].

Làm gì khi cả hai số có độ dài m và n , $m \neq n$? Để xác định, giả sử rằng $m < n$. Nếu chúng ta điền số ngắn hơn bằng các số 0 ở đầu, chúng ta sẽ nhận được một thuật toán có độ phức tạp $\Theta(n^{\log_2 3})$ so với $\Theta(mn)$ trong thuật toán cổ điển. Từ đó có thể dễ dàng thấy rằng thuật toán cổ điển tốt hơn so với các cải tiến ở $m < n^{(\log_2 3)-1}$. Một giải pháp hay là chia các chữ số của số khối có độ dài lớn hơn m . Mỗi khối này được nhân với số đầu tiên bằng cách sử dụng thuật toán cải tiến, sau đó với sự trợ giúp của $\lceil n/m \rceil$ các phép cộng và hiệu số bổ sung, kết quả cuối cùng sẽ thu được. Do đó, tổng độ phức tạp của thuật toán được sửa đổi là $\Theta(mn^{(\log_2 3)-1})$.

Câu hỏi logic được đặt ra: Nếu thuật toán này thực sự tốt như vậy, tại sao chúng ta không học nó ở trường? Vì hai lý do: thứ nhất, nó phức tạp hơn để giải thích và khó áp dụng hơn nhiều, đặc biệt là đối với những con số dài hơn. Khái niệm "đệ quy" là trừu tượng và khó hiểu. Mặt khác, có tính đến các hàng số ẩn, chúng ta có thể mong đợi rằng thuật toán cải tiến tốt hơn thuật toán cổ điển cho các số có ít nhất 500 chữ số. Hầu như không ai thực hiện các phép tính thủ công như vậy ở trường...

Bài tập

- ▷ 7.22. Hãy thực hiện thuật toán được đề xuất để nhân nhanh các số dài.
- ▷ 7.23. Hãy xác định bằng thực nghiệm, thuật toán nhân các số dài tốt hơn thuật toán cổ điển có bao nhiêu chữ số.
- ▷ 7.24. Hãy cải thiện thuật toán nhân nhanh các số dài, vì mục đích này, các số được chia thành ba thay vì hai phần. Chứng tỏ rằng 5 (thay vì 9) phép nhân với các số có độ dài $n/3$ là đủ để tính tích. Xác định độ phức tạp của thuật toán kết quả.
- ▷ 7.25. Dựa vào bài toán trước để chứng tỏ rằng phép nhân $2k - 1$ là đủ để tính tích trong trường hợp ngắn nhôm gồm k chữ số liên tiếp. Chứng tỏ rằng do đó có sự tồn tại của thuật toán nhân các số dài có độ phức tạp $\Theta(n^x)$ với mọi $x > 1$.

7.9. Bài toán tháp Hà Nội

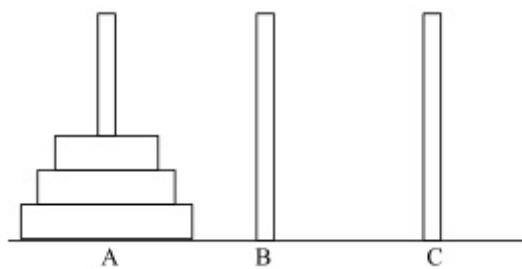
Bài toán: Có n số đĩa có đường kính khác nhau và ba trụ A, B, C . Các đĩa được xâu vào trụ thứ nhất theo thứ tự giảm dần và tạo thành một hình tháp. (xem Hình 7.8) Chúng phải được chuyển từ trụ đầu tiên sang trụ cuối cùng tuân theo các quy tắc sau:

1. Mỗi lượt có thể di chuyển một đĩa và đĩa này phải là đầu của một trong các trụ.
2. Đĩa có đường kính lớn hơn không được xếp chồng lên đĩa có đường kính nhỏ hơn.

Chúng ta sẽ trình bày biến thể đệ quy cổ điển dựa trên phương pháp chia để trị, cũng như một số giải pháp lắp thú vị nhất. Đối với những độc giả tỏ ra quan tâm đến vấn đề này, chúng ta khuyên bạn nên làm quen với [Banchev-1993], nơi một số thuật toán khác được chỉ ra.

Thuật toán 1

Đây là thuật toán cổ điển để giải quyết vấn đề. Ý tưởng là để di chuyển n đĩa từ cột A sang cột C , cần phải di chuyển $n - 1$ đĩa từ cột A sang cột B , sau đó di chuyển đĩa có số n (là lớn nhất trong số họ) từ cột A sang cột C và cuối cùng là di chuyển $n - 1$ đĩa còn lại từ cột



Hình 7.8. Tháp Hà Nội ban đầu

B sang cột *C*. Do đó, chúng ta tự nhiên giảm vấn đề thành giải hai bản sao nhỏ hơn khác của nó (chia), có liên kết, cùng với một phép toán bổ sung (chuyển đĩa lớn nhất) cho chúng ta giải pháp mong muốn (quy tắc). Việc triển khai thuật toán này trông giống như sau:

Chương trình 7.25. Bài toán tháp Hà Nội (725hanoi.c)

```
#include <stdio.h>
const unsigned n = 4;

void diskMove(unsigned n, char a, char b)
{ printf("Di chuyển đĩa %u từ %c sang %c.\n", n, a, b); }

void hanoi(char a, char c, char b, unsigned numb)
{ if (1 == numb) diskMove(1, a, c);
  else {
    hanoi(a, b, c, numb - 1);
    diskMove(numb, a, c);
    hanoi(b, c, a, numb - 1);
  }
}

int main()
{ printf("Số đĩa: %u\n", n);
  hanoy('A', 'C', 'B', n);
  return 0;
}
```

Sau đây là kết quả mẫu từ việc thực hiện chương trình cho $n = 2, 3$ và 4:

Số đĩa: 2

Di chuyển đĩa 1 từ A đến B.

Di chuyển đĩa 2 từ A đến C.

Di chuyển đĩa 1 từ B đến C.

Số đĩa: 3

Di chuyển đĩa 1 từ A đến C.

Di chuyển đĩa 2 từ A đến B.

Di chuyển đĩa 1 từ C sang B.

Di chuyển đĩa 3 từ A đến C.

Di chuyển đĩa 1 từ B đến A.

Di chuyển đĩa 2 từ B đến C.

Di chuyển đĩa 1 từ A đến C.

Số đĩa: 4

Di chuyển đĩa 1 từ A đến B.

Di chuyển đĩa 2 từ A đến C.

Di chuyển đĩa 1 từ B đến C.

Di chuyển đĩa 3 từ A đến B.

Di chuyển đĩa 1 từ C sang A.

Di chuyển đĩa 2 từ C sang B.

Di chuyển đĩa 1 từ A đến B.

Di chuyển đĩa 4 từ A đến C.

Di chuyển đĩa 1 từ B đến C.

Di chuyển đĩa 2 từ B đến A.

Di chuyển đĩa 1 từ C sang A.

Di chuyển đĩa 3 từ B đến C.

Di chuyển đĩa 1 từ A đến B.

Di chuyển đĩa 2 từ A đến C.

Di chuyển đĩa 1 từ B đến C.

Đối với một bài toán nhất định, có nhiều cách khác nhau để chuyển đổi một giải pháp đệ quy thành một giải pháp lặp lại. Chúng ta sẽ không tập trung vào phương pháp tiêu chuẩn để chuyển đổi đệ quy thành lặp lại bằng cách sử dụng ngăn xếp, mà sẽ chỉ xem xét các thuật toán đủ rõ ràng dựa trên một bộ quy tắc đảm bảo rằng có thể có một bước di chuyển duy nhất ở mỗi bước. Chúng ta sẽ để người đọc tự thấy rằng tất cả các thuật toán sau đây đều xây dựng một giải pháp giống như thuật toán 1. Trên thực tế, nó là một phát biểu khác về cùng một ý tưởng.

Thuật toán 2

Việc xây dựng giải pháp được xác định bởi các quy tắc sau:

1. Nếu n lẻ, mỗi đĩa có số lẻ luôn được chuyển sang phải và những đĩa có số chẵn luôn được chuyển sang trái. Nếu n chẵn thì - chiều ngược lại.

2. Không có hai lần chuyển động liên tiếp nào được thực hiện cùng một đĩa.

3. Không đặt đĩa lớn hơn trên đĩa nhỏ hơn.

Thuật toán 3

Quy tắc:

1. Đối với các lần di chuyển số lẻ, đĩa nhỏ nhất di chuyển sang phải hoặc trái, tùy thuộc vào việc n là lẻ (trái) hay chẵn (phải).

2. Trong trường hợp chuyển động là số chẵn, chuyển động duy nhất có thể được thực hiện trong đó đĩa nhỏ nhất không tham gia.

3. Không đặt đĩa lớn hơn trên đĩa nhỏ hơn.

Thuật toán 4

Một trong các cột được chọn là đã đánh dấu và hướng thay đổi của cột được đánh dấu được cố định. Tại n lẻ, C được đánh dấu ban đầu và hướng thay đổi là sang trái; đối với n chẵn B được đánh dấu và hướng thay đổi được đảo ngược.

1. Nếu có các đĩa trên hai trụ không được đánh dấu, di chuyển đĩa nhỏ hơn trong hai đĩa trên cùng sang đĩa lớn hơn. Nếu một trong các cột chưa được đánh dấu trống, đĩa trên cùng của cột kia sẽ được di chuyển trên đó. Nếu cái kia trống, thì bài toán đã được giải quyết.

2. Vị trí tiếp giáp với cực được đánh dấu theo hướng thay đổi đã được cố định trước được chọn để đánh dấu.

Bài tập

▷ 7.26. Hãy triển khai thực hiện các thuật toán 2, 3 và 4.

▷ 7.27. Chứng minh rằng các thuật toán 2, 3 và 4 thực hiện các hành động tương tự như thuật toán 1.

7.10. Tổ chức giải vô địch bóng đá

Một bài toán kinh điển khác, được giải quyết một cách hiệu quả với sự giúp đỡ của sự phân chia để trị, là chuẩn bị một kế hoạch giải đấu kiểu "mọi người chống lại mọi người". Ví dụ:

Giải vô địch bóng đá . Bộ Văn Hóa và Thể thao của một nước ASEAN phải tổ chức một giải vô địch bóng đá với n đội (đánh số từ 1 đến n). Chức vô địch phải được tổ chức trong n vòng nếu n lẻ, và $n - 1$ vòng nếu n là chẵn. Cứ hai đội thi đấu với nhau đúng một lần. Tạo một chương trình quản lý chương trình của giải vô địch.

Kết quả phải là một bảng gồm n hàng. Số từ cột thứ j của hàng thứ i của bảng có nghĩa là số của đội đấu với đội có số i trong vòng tròn số j . (Nếu đội i đấu với đội k ở vòng j , điều này đương nhiên có nghĩa là đội k đấu với đội i trong cùng một vòng.)

Nếu đội thứ i nghỉ ở vòng thứ j , thì số thứ j ở hàng thứ i là 0.

Ví dụ: đối với $n = 3$ và $n = 4$, lịch trình tương ứng sẽ như thế này (dọc: đội, ngang: vòng):

2	3	0	2	3	4
1	0	3	1	4	3
0	1	2	4	1	2
3	2	1			

Lời giải: Chúng ta sẽ cho phép mình thay đổi một chút hình dạng của kết quả. Thay vì bảng được chỉ định trong điều kiện, chúng ta sẽ tạo một bảng khác, luôn có kích thước $n \times n$ (thậm chí đối với n chẵn), thuộc loại "đội-nhóm", như trên hàng thứ i và cột thứ j sẽ là vòng tròn, trong đó sẽ gấp các đội có số thứ tự i và j ($1 \leq i, j \leq n$). Chúng ta để việc chuyển đổi bảng sang định dạng yêu cầu trong bài toán cho người đọc như một bài tập dễ dàng.

Rõ ràng bài toán không có ý nghĩa gì ở $n = 1$. Tại $n = 2$, đúng một trò chơi được chơi, mất đúng một ngày: xem Bảng 7.1.

Chúng ta giới thiệu ký hiệu viết tắt $i : j$, có nghĩa là "đội ta gấp đội j " ($1 \leq i, j \leq n$). Trong trường hợp bốn đội, giải đấu kết thúc sau 3 ngày. Làm thế nào để xây dựng lịch trình cuộc họp phù hợp? Chúng ta có thể chia các đội thành các cặp, như trong ngày đầu tiên

Đội	1	2
1	0	1
2	1	0

Bảng 7.1. Lịch thi đấu của 2 đội

1: 2 và 3: 4, vào ngày thứ hai 1: 3 và 2: 4, và vào ngày thứ ba - 1: 4 và 2: 3. Ta được Bảng 7.2. Khi xem xét kỹ hơn, chúng ta nhận thấy rằng

Đội	1	2	3	4
1	0	1	2	3
2	1	0	3	2
3	2	3	0	1
4	3	2	1	0

Bảng 7.2. Lịch thi đấu của 4 đội

bảng này chứa gấp đôi bảng cho $n = 2$ và hai lần bảng

$$\begin{matrix} 2 & 3 \\ 3 & 2 \end{matrix}$$

Lưu ý rằng sau này thu được từ

$$\begin{matrix} 0 & 1 \\ 1 & 0 \end{matrix}$$

bằng cách thêm phần tử 2.

Bạn đã biết làm thế nào chúng ta có thể có được kế hoạch giải đấu tương ứng cho $n = 8$? Sao chép bảng cho $n = 4$ bốn lần, sau đó thêm từng phần tử số 4 vào các bảng có kích thước 4 bên ngoài đường chéo chính. (xem Bảng 7.3)

Bảng 7.3 được xác minh trực tiếp. là một lịch thi đấu thực sự: mỗi hàng và mỗi cột chứa một hoán vị của các số từ 0 đến 7 mà không lặp lại, tức là giải đấu kết thúc sau 7 ngày, với mỗi đội thi đấu đúng

Đội	1	2	3	4	5	6	7	8
1	0	1	2	3	4	5	6	7
2	1	0	3	2	5	4	7	6
3	2	3	0	1	6	7	4	5
4	3	2	1	0	7	6	5	4
5	4	5	6	7	0	1	2	3
6	5	4	7	6	1	0	3	2
7	6	7	4	5	2	3	0	1
8	7	6	5	4	3	2	1	0

Bảng 7.3. Lịch thi đấu của 8 đội.

một trận mỗi ngày. Bảng là đối xứng, tức là nếu $i : j$ thì $j : i$ đúng, và ngược lại ($1 \leq i, j \leq n$). Có các số không trên đường chéo chính, tức là $i : i$ không đúng với $1 \leq i \leq n$. Theo cách tương tự, chúng ta nhận được bảng cho $n = 16, 32, 64, \dots$ và nói chung cho n , bậc chính xác là 2. Điều này dễ dàng theo sau bằng quy nạp. Chúng ta để lại phần chứng minh cho người đọc như một bài tập dễ dàng. Hãy suy nghĩ lại. Cho $n = 2^k$. Để có một bảng cho n , chúng ta cần lấy một bảng cho $n/2$ (phép chia), sau đó sao chép nó 4 lần và thêm một hàng số vào hai trong số các bản sao (quy tắc). Sau đây là một ví dụ triển khai thuật toán được mô tả.

Chương trình 7.26. Lịch thi đấu giải vô địch(726tourn1.c)

```
#include <stdio.h>
#define MAX 100
unsigned m[MAX][MAX];

void copyMatrix(unsigned stX, unsigned stY, unsigned cnt, unsigned add)
{
    unsigned i, j;
    for (i = 0; i < cnt; i++)
        for (j = 0; j < cnt; j++)
            m[i + stX][j + stY] = m[i + 1][j + 1] + add;
}
```

```

void findSolution(unsigned n) /*Xây dựng bảng*/
{
    unsigned i;
    m[1][1] = 0;
    for (i = 1; i <= n; i <<= 1) {
        copyMatrix(i + 1, 1, i, i);
        copyMatrix(i + 1, i + 1, i, 0);
        copyMatrix(1, i + 1, i, i);
    }
}

void print(unsigned n) /* In ra kết quả*/
{
    unsigned i, j;
    for (i = 1; i <= n; i++) {
        for (j = 1; j <= n; j++)
            printf("%3u", m[i][j]);
        printf("\n");
    }
}

int main() {
    const unsigned n = 8;
    findSolution(n);
    print(n);
    return 0;
}

```

Thật không may, ở dạng này, thuật toán không phù hợp với n , không phải là lũy thừa của 2. Chúng ta để người đọc cố gắng sửa đổi để nó hoạt động với mọi n tự nhiên. Dưới đây chúng ta sẽ xem xét một thuật toán khác hợp lệ cho mỗi n .

Đối với n bảng lẻ, chúng ta sẽ tạo ra như sau: Chúng ta sẽ không điền vào bảng $n \times n$, mà là một bảng với một cột bổ sung khác, tức là $n \times (n + 1)$. Chúng ta sẽ điền vào các hàng, bắt đầu từ cột ngoài cùng bên trái và di chuyển sang bên phải. Khi chúng ta điền vào cột $(n + 1)$ của hàng hiện tại, chúng ta sẽ chuyển sang hàng tiếp theo. Chúng ta sẽ bắt đầu điền với giá trị 1 trong hàng 1 của cột 1, và chúng ta sẽ điền vào ô tiếp theo với giá trị tiếp theo trong khoảng $[1; n]$. Khi đạt đến giá trị của n , chúng ta sẽ bắt đầu lại từ 1. Bảng 7.4. hiển thị kết quả tại $n = 5$. Tuy nhiên, có thể thấy rằng chúng ta có

thể không điền vào cột thứ $(n + 1)$ - nó trùng với cột đầu tiên. (Tại sao?)

Đội	1	2	3	4	5	6
1	1	2	3	4	5	1
2	2	3	4	5	1	2
3	3	4	5	1	2	3
4	4	5	1	2	3	4
5	5	1	2	3	4	5

Bảng 7.4. Bắt đầu điền lịch đấu cho 5 đội.

Sau khi điền xong, chúng ta đặt lại tất cả các phần tử trên đường chéo chính của ma trận (xem Bảng 7.5). Với thậm chí n , chúng ta chỉ

Đội	1	2	3	4	5	6
1	0	2	3	4	5	1
2	2	0	4	5	1	2
3	3	4	0	1	2	3
4	4	5	1	0	3	4
5	5	1	2	3	0	5

Bảng 7.5. Lịch thi đấu của 5 đội.

có thể giới thiệu một đội hư cấu. Tuy nhiên, ở đây, chúng ta sẽ áp dụng một chiến lược khác. Chúng ta sẽ giải bài toán cho $n - 1$, là số lẻ, sau đó, trong khi đặt lại đường chéo chính, chúng ta sẽ điền thêm vào hàng và cột thứ n : đội thứ k phải chơi với đội thứ n , vì không có n người đi nghỉ.

Ví dụ, để lập lịch trình cho $n = 4$, trước tiên chúng ta điền vào bảng cho $n = 3$ (xem Bảng 7.6).

Sau đó, chúng ta sẽ đặt lại đường chéo chính và điền vào hàng thứ n như trong Bảng 7.7. Sự thay đổi tương ứng của hàm `findSolution()` như sau

Đội	1	2	3	4
1	1	2	3	1
2	2	3	1	2
3	3	1	2	3

Bảng 7.6. Bắt đầu điền lịch họp cho 3 đội.

Đội	1	2	3	4
1	0	2	3	1
2	2	0	1	2
3	3	1	0	3
4	1	2	3	0

Bảng 7.7. Lịch thi đấu của 4 đội.

Chương trình 7.27. Sửa đổi hàm findSolution() (727tourn2.c)

```

void findSolution(unsigned n) /* Thiết lập bảng */
{
    unsigned i;
    unsigned saveN = n;
    if (n % 2 == 0) /* Nếu n chẵn bài toán đưa về n-1 */
        n--;
    /* Điền vào bảng n n - ở đây đảm bảo số lẻ. */
    for (i = 0; i < n * (n + 1); i++)
        m[i % (n + 1)][i / (n + 1)] = i % n + 1;
    /* Phục hồi với giá trị của n */
    n = saveN;

    for (i = 0; i < n; i++) {
        if (n%2 == 0) /* điền vào cột và hàng cuối cùng số chẵn n */
            m[i][n - 1] = m[n - 1][i] = m[i][i];
        m[i][i] = 0; /* Đường chéo chính điền với 0 */
    }
}

```

Giải pháp trên, mặc dù hiệu quả, nhưng không cho phép trả lời trực tiếp cho câu hỏi: Đội i và đội j gặp nhau trong vòng tròn nào? Điều này yêu cầu hoàn thành đầy đủ hoặc ít nhất một phần của

bảng liên quan. Khi xem xét kỹ hơn các bảng trên, có thể rút ra các công thức thích hợp.

Hãy có một số người tham gia chẵn ($n = 2k$). Cho người tham gia A và B có số s và t khác n . Sau đó chơi trong vòng tròn $s + t - 1$ nếu $s + t \leq n$, và trong $s + t - n$ nếu $s + t > n$. Trong một giải đấu cờ vua, ai chơi với quân trắng và ai - với quân đen cũng rất quan trọng. Một giải pháp khả thi là tuân theo quy tắc: người chơi có số nhỏ hơn sẽ chỉ chơi với người quân trắng nếu $s + t$ là số lẻ. Người có số n chơi với người có số l trong vòng tròn $2l - 1$, nếu $2l \leq n$ và trong vòng $2l - n$, khi $2l > n$. Trong một giải đấu cờ vua có số người tham gia từ 1 đến $n/2$, anh ta chơi với quân đen và những người khác với quân trắng. Trong trường hợp có số lượng người tham gia lẻ, chúng ta thêm một người tham gia hư cấu và người chơi cùng anh ta sẽ ở trong vòng kết nối tương ứng.

Mặc dù sơ đồ sau phù hợp hơn để có được câu trả lời trực tiếp cho câu hỏi " i và j đóng trong vòng tròn nào?", Nhưng nó cũng có thể được sử dụng để điền vào bảng do tính đơn giản của công thức. Một lần nữa, chúng ta có thể từ bỏ nhóm hư cấu và điền trực tiếp vào các giá trị tương ứng của trụ cuối cùng. Đây là cách chúng ta nhận được:

Chương trình 7.28. Sửa đổi hàm findSolution() (728tourn3.c)

```
void findSolution(unsigned n) /* Thiết lập bảng */
{
    unsigned i, j;
    unsigned saveN = n;
    if (n % 2 == 0) /* nếu n chẵn, bài toán đưa về n-1 */
        n--;
    /* Điền đầy bảng cho n - đây đảm bảo số lẻ. */
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if ((m[i][j] = i + j + 1) > n)
                m[i][j] -= n;

    /* Khôi phục giá trị của n */
    n = saveN;
    for (i = 0; i < n; i++) {
        if (n % 2 == 0) /* điền cột và hàng cuối, số chẵn n */

```

```

m[i][n - 1] = m[n - 1][i] = m[i][i];
m[i][i] = 0; /* Đèn đường chéo chính bằng 0 */
}
}

```

Hàm trên khác một chút so với cách lập luận lý thuyết bởi vì chỉ mục của mảng trong C bắt đầu từ 0 thay vì 1.

Bài tập

- ▷ 7.28. Hãy chuyển đổi bảng đã xây dựng thành một bảng thuộc loại được yêu cầu trong điều kiện của bài toán từ Giải thể thao vô địch Việt Nam.
- ▷ 7.29. Hãy chứng minh tính đúng đắn của thuật toán đầu tiên để biên soạn lịch thi đấu cho n bậc 2.
- ▷ 7.30. Hãy thực hiện thuật toán thứ hai để biên soạn lịch thi đấu, giới thiệu một đội giả tưởng cho n chẵn.
- ▷ 7.31. Hãy chứng minh tính đúng đắn của thuật toán thứ hai để biên soạn lịch thi đấu cho mỗi n tự nhiên, $n \neq 2$.
- ▷ 7.32. Hãy suy ra các công thức đã cho để lập lịch thi đấu cờ vua.

7.11. Sự dịch chuyển tuần hoàn của các phần tử mảng

Vào những năm 1960, Carnigan và Plodger đã gặp phải vấn đề chuyển vị theo chu kỳ các phần tử mảng khi phát triển một trình soạn thảo văn bản cho hệ thống UNIX. Khả năng tính toán khiêm tốn của các máy tính thời đó đặt ra các yêu cầu nghiêm ngặt về hiệu quả của hoạt động này cả về thời gian sử dụng của CPU và RAM cần thiết. Phiên bản gốc của chương trình, dựa trên các danh sách được liên kết, được chứng minh là không hiệu quả và phức tạp, thậm chí còn chứa một số lỗi. Sau khi xem xét kỹ hơn bản chất của bài toán, cả hai đã cố gắng tạo ra một chương trình ngắn, hiệu quả cả về thời gian và bộ nhớ, sau này được sử dụng trong một số trình soạn thảo văn bản khác.

Có tổng cộng năm thuật toán khác nhau để giải quyết vấn đề, tất cả chúng sẽ được thảo luận bên dưới. Hãy để chúng ta ký hiệu mảng bằng m và số phần tử của nó bằng n . Làm thế nào chúng ta có thể thay đổi? Có hai giải pháp khá đơn giản, mỗi giải pháp đều có nhược điểm của nó: thứ nhất là không hiệu quả về bộ nhớ và thứ hai là về mặt thời gian.

Hãy xem giải pháp đầu tiên. Nếu chúng ta có một lượng RAM bổ sung nhất định, chúng ta có thể thực hiện như sau

Thuật toán 1

1. Sao chép k phần tử đầu tiên của m vào mảng thời gian x nào đó.
2. Chuyển $n - k$ phần tử còn lại sang bên trái k vị trí.
3. Sao chép các phần tử từ x trở lại m (ở k vị trí cuối cùng).

Thuật toán trên rõ ràng là không hiệu quả về dung lượng RAM được sử dụng. Đồng thời, rõ ràng là trong trường hợp đặc biệt tại $k = 1$, chúng ta có thể thực hiện hàm `shiftBy1()`, hàm này sẽ yêu cầu bộ nhớ bổ sung liên tục và thời gian có bậc là $\Theta(n)$. Chúng ta nhận được một thuật toán mới:

Thuật toán 2

1. Gọi k lần `shiftBy1()`.

Thật không may, thuật toán được đề xuất, mặc dù hiệu quả trong bộ nhớ, trong trường hợp xấu nhất đòi hỏi thời gian có bậc là $\Theta(n^2)$.

Chẳng lẽ chúng ta không thể tìm ra một thuật toán vừa tiết kiệm thời gian vừa tiết kiệm bộ nhớ, kết hợp những ưu điểm của các thuật toán trên? Nó chỉ ra rằng một thuật toán như vậy tồn tại. Ý tưởng là chuyển phần tử đầu tiên của $m[]$ sang một biến tạm thời `tmp`, sau đó chuyển $m[k + 1]$ thành $m[1]$, $m[2k + 1]$ thành $m[k + 1]$, v.v., lấy tất cả các chỉ số của $m[]$ modulo n . Quá trình tiếp tục cho đến khi đạt đến $m[1]$, trong đó `tmp` được sử dụng thay thế. Trong trường hợp không phải tất cả các phần tử đều đúng vị trí, chúng ta lặp lại quy trình, bắt đầu bằng $m[2]$, v.v. Một ví dụ về triển khai ý tưởng đã trình bày trông như thế này (Ở đây, các chỉ mục được thay đổi vì lập chỉ mục của mảng bắt đầu từ 0):

Chương trình 7.29. Dịch chuyển tuần hoàn của các phần tử mảng (729shift1.c)

```
#include <stdio.h>
#define MAX 100

struct CElem {
    int data;
    /* có thể thêm... */
} m[MAX];

const unsigned n = 10; /* Số phần tử trong mảng */
const unsigned k = 2; /* Số vị trí di chuyển*/

void init(void)
{ unsigned i;
    for (i = 0; i < n; i++)
        m[i].data = i;
}

unsigned gcd(unsigned x, unsigned y)
{ while (y > 0) {
    unsigned tmp = y;
    y = x % y;
    x = tmp;
}
    return x;
}

void shiftLeft1(unsigned k)
{ /* di chuyển m[] với k vị trí từ trái, sử dụng một biến phụ trợ*/
    unsigned i, ths, next, gcdNK;
    struct CElem tmp;
    for (gcdNK = gcd(n, k), i = 0; i < gcdNK; i++) {
        ths = i; tmp = m[ths];
        next = ths + k;
        if (next >= n)
            next -= n;
        while (next != i) {
            m[ths] = m[next];
            ths = next;
            next += k;
        }
    }
}
```

```

        next += k;
        if (next >= n) next -= n;
    }
    m[ths] = tmp;
}
}

void print(void)
{ unsigned i;
    for (i = 0; i < n; i++)
        printf("%d ", m[i].data);
    printf("\n");
}

int main()
{
    init();
    shiftLeft1(k);
    print();
    return 0;
}

```

Chúng ta có thể nhìn mọi thứ theo một cách khác. Gọi A là k phần tử đầu tiên của m , và B là $n - k$ còn lại. Khi đó, phép di chuyển tuần hoàn các phần tử của dãy m sang trái tại k vị trí có thể coi là một phép biến hình AB thành BA . Không giới hạn cộng đồng lý luận, chúng ta chắc chắn có thể coi rằng $k < n/2$, tức là A ngắn hơn B . Chúng ta chia B thành hai phần B_L và B_R , vì B_R chứa k phần tử. Nhiều nhất là A . Sau đó, sự chuyển dịch tuần hoàn giảm đến sự biến đổi AB_LB_R thành B_LB_RA . Nếu chúng ta hoán đổi vị trí của A và B_R (mà k trao đổi là đủ cho chúng ta), các phần tử của A sẽ đến vị trí cuối cùng của chúng. Nó vẫn là để giải quyết vấn đề với B_L và B_R - cả hai phần của B . Đối với B , chúng ta nhận được một bài toán mới, tương tự như ban đầu, nhưng với kích thước nhỏ hơn. Chúng ta có thể tiếp tục quy trình một cách đệ quy, cho đến khi giải pháp cuối cùng của nhiệm vụ: chia và trị. Mặc dù thuật nhìn có vẻ phức tạp nhưng quy trình được mô tả đủ đơn giản và có thể được mô tả lặp đi lặp lại. Grizzly và Mills cung cấp nhận thức lặp đi lặp lại sau:

Chương trình 7.30. Dịch chuyển tuần hoàn của các phần tử mảng (730shift2.c)

```

void swap(unsigned a, unsigned b, unsigned l)
{ /* Thay chỗ mảng con m[a..a+l-1] và m[b..b+l-1] */
    unsigned i;
    struct CElem tmp;
    for (i = 0; i < l; i++)
    {
        tmp = m[a + i];
        m[a + i] = m[b + i];
        m[b + i] = tmp;
    }
}

void shiftLeft2(unsigned k)
{ /* Di chuyển mảng m[] bằng k vị trí từ trái.
   * quy trình đệ quy được thực hiện lặp đi lặp lại */
    unsigned i, j, p;
    p = i = k;
    j = n - k;
    while (i != j)
        if (i > j) { swap(p - i, p, j); i -= j; }
        else { swap(p - i, p + j - i, i); j -= i; }
    swap(p - i, p, i);
}

```

Vì vậy, bây giờ chúng ta có hai thuật toán hiệu quả về thời gian và hiệu quả về bộ nhớ. Chúng ta sẽ đề xuất một thuật toán hiệu quả và đặc biệt phức tạp khác, cụ thể là thuật toán được sử dụng bởi Carnigan và Plodger, dựa trên nhận dạng toán học đơn giản sau:

$$BA = (A^R B^R)^R.$$

Với A^R ở trên ta ký hiệu mảng thu được từ các phần tử của A , được lấy theo thứ tự ngược lại. Ý nghĩa của B^R và $(A^R B^R)^R$ là tương tự. Sự đồng nhất ở trên cho chúng ta một thuật toán cực kỳ đơn giản và hiệu quả để giải quyết vấn đề, đó là: để có được BA , chúng ta cần đảo ngược A , đảo ngược B , và sau đó đảo toàn bộ mảng. Sau đây là hiện thực của thuật toán được trình bày:

Chương trình 7.31. Dịch chuyển tuần hoàn của các phần tử mảng (731shift3.c)

```

void reverse(unsigned a, unsigned b) /* Đảo ngược mảng con m[a..b]
{
    unsigned i, j, k, cnt;
    struct CElem tmp;
    for (cnt = (b-a)/2, k=a, j=b, i=0; i <= cnt; i++, j--, k++) {
        tmp = m[k];
        m[k] = m[j];
        m[j] = tmp;
    }
}

void shiftLeft3(unsigned k)
{
    /* Di chuyển mảng m đến k vị trí sang trái, trong ba bước*/
    reverse(0, k - 1);
    reverse(k, n - 1);
    reverse(0, n - 1);
}

```

Ba thuật toán cuối cùng yêu cầu bộ nhớ bổ sung liên tục và được thực thi trong thời gian $\Theta(n)$. Tuy nhiên, ở các giá trị cao hơn của n , chúng không tương đương về tốc độ, vì hằng số trước n khác nhau. Thật vậy, với ShiftLeft1() tương đối phức tạp, mỗi phần tử của mảng được đọc và ghi nhớ chính xác một lần, trong khi với ShiftLeft3() các phép toán này dường như được thực hiện hai lần (xem [Bentley-1990], [Nakov-1998d]).

Bài tập

- ▷ 7.33. Hãy so sánh về mặt lý thuyết thời gian hoạt động của các hàm shiftLeft1(), shiftLeft2() và shiftLeft3().
- ▷ 7.34. Hãy suy ra công thức $BA = (A^R B^R)^R$.

7.12. Câu hỏi và bài tập

- ▷ 7.35. *Sắp xếp nhanh tốt hơn*

Sử dụng thuật toán trung vị trung bình (xem 7.2), Sắp xếp lại sắp xếp nhanh (xem 3.1.6.) Để nó luôn chạy trong thời gian $\Theta(n \log_2 n)$. So sánh độ phức tạp lý thuyết và thực nghiệm với độ phức tạp của

phân loại hình chopy (xem 3.1.9.).

▷ 7.36. *k-phần tử nhỏ nhất trong một mảng*

Hãy phát triển một thuật toán để tìm k số nhỏ nhất trong một mảng. So sánh các thuật toán sau:

- sắp xếp và hiển thị k số đầu tiên
- tìm phần tử x lớn nhất thứ k bằng thuật toán tuyển tính (xem 7.1.), Phân chia mảng phân vùng trái và phải đối với x và sắp xếp hoàn chỉnh phân vùng bên trái
- xây dựng một kim tự tháp (hàng đợi ưu tiên) và chiết xuất gấp k của mức tối thiểu phần tử hàng đầu của nó.

▷ 7.37. *Cặp điểm gần nhất*

Có n điểm trong mặt phẳng được cho bởi tọa độ của chúng. Tìm một cặp điểm mà khoảng cách giữa hai điểm là nhỏ nhất. Thuật toán của bạn có hoạt động trong không gian 3D không? Và trong n -chiều?

▷ 7.38. *Vỏ lồi ở nhiều điểm*

Có n điểm trong mặt phẳng được cho bởi tọa độ của chúng. Tìm một đa giác lồi có một mặt nhỏ nhất (tập lồi nhỏ nhất) có chứa chúng. Thuật toán của bạn có hoạt động trong không gian 3D không? Và trong n -chiều?

▷ 7.39. *Các ví dụ khác về chia và chinh phục*

Có đúng là các thuật toán được liệt kê dưới đây dựa trên chia và chinh phục không? Tại sao?

- Phân loại Hoor nhanh (xem 3.1.6.)
- phân loại theo hình chopy (xem 3.1.9.), Một phần (tại sao?)
- sắp xếp bit (xem 3.2.3.), Biến thể đệ quy
- tìm kiếm nhị phân (xem 4.3.)
- Tìm kiếm Fibonacci (xem 4.4.)
- tìm kiếm nội suy (xem 4.5.)
- Thuật toán Shannon-Fano (xem 10.4.1.)
- Thuật toán Huffman (xem 10.4.2.)

Những thuật toán nào khác được coi là có thể được thêm vào?

DANH SÁCH CÁC HÌNH

1.1	Sơ đồ Cen	6
1.2	Tập hợp con (a), tập hợp hợp (b), tập giao (c) và tập hiệu (d) của các tập hợp.	6
1.3	Ma trận $m \times n$	19
1.4	Thu thập thông tin các phần tử của ma trận: (a) theo hàng và (b) theo cột.	20
1.5	Tổng các ma trận.	21
1.6	Eratosthenes và lưới của mình	29
1.7	$2^{11213} - 1$ là số nguyên tố	37
1.8	Tam giác Pascal	43
1.9	Số Fibonacci trong kiến trúc và nghệ thuật	62
1.10	Leonardo Fibonacci.	63
1.11	Số Fibonacci trong tự nhiên.	64
1.12	Bài toán những con thỏ.	65
1.13	Mối quan hệ của hai số Fibonacci liên tiếp.	66
1.14	Cây gọi hàm đệ quy.	67
1.15	Tam giác Stirling	104
1.16	Kim tự tháp loại 1	145
1.17	Kim tự tháp loại 2	146
1.18	Quảng trường	146
1.19	Hình vuông ma thuật	147
1.20	Xây dựng một hình vuông kỳ diệu.	147
1.21	Hình vuông La tinh	149
1.22	Vòng cổ	150

2.1	Ngăn xếp	159
2.2	Hàng đợi	160
2.3	Danh sách chia bộ nhớ	162
2.4	Danh sách chia bộ nhớ	165
2.5	Hàng đợi thể hiện quả mảng chu kỳ	165
2.6	Danh sách liên kết động	168
2.7	Đưa vào đầu và dịch chuyển con trỏ	170
2.8	Đưa vào sau phần tử chỉ ra từ bảng chỉ dẫn	172
2.9	Loại trừ khỏi danh sách.	173
2.10	Biểu diễn trực quan thông qua các tập hợp lồng nhau.	179
2.11	Trình bày trong ngoặc đơn.	179
2.12	Biểu diễn bằng đồ thị liên thông xoay chiều có định hướng (xem Chương 5).	180
2.13	Biểu diễn bằng một đồ thị liên thông xoay chiều không có hướng (xem Chương 5), với một trong các đỉnh được chọn làm gốc: trong trường hợp này là đỉnh A.	180
2.14	Biểu diễn bằng phép dời hình từng bước.	180
2.15	Cây nhị phân	181
2.16	Cây	181
2.17	Trình bày biểu thức $a + ((c * d) * (e^b))$	181
2.18	Trình bày động của một cây nhị phân trong bộ nhớ.	182
2.19	Cây tìm kiếm nhị phân.	183
2.20	Xóa kết nối đầu bằng từ khóa 5.	185
2.21	Xóa kết nối đầu bằng từ khóa 10.	185
2.22	Trường hợp xấu nhất với cây nhị phân.	192
2.23	Cây Fibonacci của hàng 0.	193
2.24	Cây Fibonacci của hàng 1.	193
2.25	Cây Fibonacci của hàng 2.	194

3.1	Cây so sánh cho tập $\{a, b, c\}$.	235
3.2	Sắp xếp theo cách chèn.	237
3.3	Lược đồ sắp xếp để phân loại chèn.	298
3.4	Lược đồ sắp xếp chǎn-lẻ.	299

5.1	Đồ thị định hướng	332
5.2	Đồ thị cân định hướng	332
5.3	Đồ thị không định hướng	333
5.4	Bản đồ giao thông thành phố Hà Nội	334
5.5	Được quy nạp bởi tập $V' = \{1, 2, 3\}$ đồ thị con G' của G	336
5.6	Đồ thị vô hướng hoàn chỉnh và trống rỗng.	337
5.7	Ma trận lân cận và đồ thị tương ứng của nó.	340
5.8	Đồ thị không định hướng.	346
5.9	Tiết kiệm một đường đi.	355
5.10	Tối ưu hóa đường.	364
5.11	Tối ưu hóa đường.	368
5.12	Đồ thị có trọng số với cạnh âm.	378
5.13	Bước đóng chuyển tiếp.	381
5.14	Tìm đường quan trọng trong đồ thị.	383
5.15	Tập hợp cơ bản của các chu trình trong đồ thị.	387
5.16	Chu trình Hamilton nhỏ nhất	393
5.17	7 cầu ở Königsberg và đồ thị của chúng	396
5.18	Chu trình Euler	397
5.19	Mối chuyển đổi	401
5.20	Luồng cực đại trong đồ thị	403
5.21	Một ví dụ về sự kém hiệu quả trong việc chọn ngẫu nhiên một đường dẫn tăng.	404

5.22 Định hướng bắc cầu	415
5.23 Giảm bắc cầu	419
5.24 Đồ thị có trọng số định hướng.	420
5.25 Sắp xếp tô pô đầy đủ	427
5.26 Điểm phân tách trong đồ thị	439
5.27 Cây bao phủ nhỏ nhất trong đồ thị	445
5.28 Đồ thị vô hướng	456
5.29 Tập đỉnh trội trong đồ thị	460
5.30 Đồ thị có hướng	464
5.31 Tâm và bán kính trong đồ thị (các cạnh có trọng số 1) . .	468
5.32 Tìm p -tâm	471
5.33 K_5	480
5.34 $K_{3,3}$	480
5.35 B -bậc của đồ thị	482
5.36 Phương pháp sóng.	487

6.1	Phân loại bài toán (theo thời gian).	491
6.2	Thỏa mãn của biểu thức boolean	504
6.3	Màu r tối thiểu của đồ thị.	508
6.4	Đường dẫn đơn dài nhất trong một đồ thị có định hướng.	512
6.5	Vị trí quân ngựa được di chuyển.	515
6.6	Lời giải bài toán với $n = 8$.	515
6.7	Đồ thị bài toán với $n = 3$ và $n = 4$	515
6.8	Lời giải bài toán quân Hậu với $n = 8$	520
6.9	Lịch dạy học	524
6.10	Đồ thị trò chơi "X" và "O"	542
6.11	Các vị trí kết thúc trong cây trò chơi	543
6.12	Chiếu tướng sau 3 nước đi (trắng đi trước)	543
6.13	Nhát cắt beta	548
6.14	Nhát cắt alpha	549

7.1	Mảng trong trường hợp $i < k$	567
7.2	Mảng trong trường hợp $j > k$	567
7.3	Mảng trong trường hợp $j < k < i$	567
7.4	Chọn điểm trung vị	571
7.5	Hợp nhất nhị phân của các mảng đã sắp xếp.	595
7.6	Động "bóc tách" của A và B .	614
7.7	Kết quả của tích của A và B .	614
7.8	Tháp Hà Nội ban đầu	621

DANH SÁCH CÁC BẢNG

1.1	Các kiểu số nguyên trong Borland C dành cho DOS.	9
1.2	Các kiểu số th trong Borland C.	10
1.3	Hệ số đa thức	43
1.4	Ghi lại một số số bằng chữ số La mã.	56
1.5	sự phụ thuộc giữa kích thước dữ liệu đầu vào và tốc độ thực thi.	106
1.6	Mối liên hệ giữa quan hệ về tiệm cận của hàm số và số thực.	116
1.7	Tăng một số hàm tiệm cận được sử dụng phổ biến hơn. .	117

- 3.1 So sánh và trao đổi trong việc sắp xếp theo cách chèn. . . 238
- 3.2 So sánh và trao đổi trong sắp xếp theo cách nổi bong bóng.244
- 3.3 So sánh và trao đổi trong sắp xếp theo cách chọn trực tiếp.260

4.1 Số phép so sánh tối đa ở các giá trị khác nhau của n và k. 315

5.1	So sánh mức độ phức tạp của các hoạt động trong các buổi biểu diễn khác nhau.	344
5.2	Độ phức tạp của việc duyệt qua một đồ thị trong các biểu diễn khác nhau.	352

5.3	Số lượng phép toán cơ sở tìm kiếm và hợp nhất các cây với n phần tử trong các chiến lược khác nhau	448
5.4	Người dịch.	460
5.5	Tâm của đồ thị: đỉnh 2	468
5.6	Ma trận trong số của đồ thị	472

6.1	Bảng chân lý	500
6.2	Các phương án lịch	525
6.3	Bảng phân công	526
6.4	Bảng phân công	526
6.5	Bảng phân công	526

7.1	Lịch thi đấu của 2 đội	625
7.2	Lịch thi đấu của 4 đội	625
7.3	Lịch thi đấu của 8 đội.	626
7.4	Bắt đầu điền lịch đấu cho 5 đội.	628
7.5	Lịch thi đấu của 5 đội.	628
7.6	Bắt đầu điền lịch họp cho 3 đội.	629
7.7	Lịch thi đấu của 4 đội.	629

Danh sách chương trình

1.1	Số chữ số (101digits.c)	11
1.2	Tính tổng cách 1 (102sum1.c)	13
1.3	Tính tổng cách 2 (103sum1.c)	13
1.4	Tính tích (104mult.c)	15
1.5	Tính lũy thừa (105power.c)	16
1.6	Giai thừa (106factorial.c)	18
1.7	Số chữ số của tích (110digtsnf.c)	23
1.8	Số chữ số (111prime.c)	26
1.9	Kiểm tra số nguyên tố (112preproc.c)	27
1.10	Danh sách số nguyên tố (113sieve.c)	30
1.11	Số nguyên tố trong một khoảng (114proc.c)	31
1.12	Phân tích số thành tích thừa số nguyên tố (115numdev.c)	34
1.13	Tìm số 0 làm phép nhân (116factzero.c)	35
1.14	Số hoàn thiện (117perfect.c)	40
1.15	Tam giác Pascal (118pascalt.c)	44
1.16	Tính hệ số Newton (119cnk.c)	45
1.17	Chuyển số thập phân thành La Mã và ngược lại (121rom2dec.c)	56
1.18	Tính giai thừa (121factrec.c)	60
1.19	Tính Fibonacci (123fibrec.c)	66
1.20	Tính Fibonacci không đệ quy (124fibiter.c)	68
1.21	Tìm ước số chung lớn nhất không đệ quy (125gcditer.c)	71
1.22	Tìm ước số chung lớn nhất đệ quy (126gcdrec.c)	71
1.23	Bội chung nhỏ nhất (127lcm.c)	73
1.24	Giá trị trả về của đệ quy (128print.c)	74
1.25	Giá trị trả về của đệ quy (129printrec.c)	75
1.26	Giá trị trả về của đệ quy (122factrec.c)	76
1.27	Giá trị trả về của đệ quy (130print1.c)	76
1.28	Giá trị trả về của đệ quy (131print2.c)	77

1.29	Giá trị trả về của đệ quy (132print3.c)	77
1.30	Tính hoán vị (133permute.c)	81
1.31	Tính hoán vị 2 (134permswap.c)	83
1.32	Số chữ số (135codeperm.c)	85
1.33	Chỉnh hợp (136variate.c)	90
1.34	Tổng bằng 0 (137sumzero.c)	92
1.35	Tìm tổ hợp (138comb.c)	95
1.36	Biểu diễn thành tổng số (139devnum.c)	98
1.37	Biểu diễn thành tích số (140devnum2.c)	100
1.38	Biểu diễn thành tích số (141devnum3.c)	101
1.39	Số phân hoạch tập hợp (142bell.c)	104

2.1	Ngăn xếp (202stack2.c)	163
2.2	Hàng đợi (203queue.c)	166
2.3	Thao tác trên danh sách (204list.c)	173
2.4	Thao tác trên cây nhị phân (205bintree.c)	185
2.5	Bảng băm (206hash.c)	218
2.6	Bảng băm tập hợp (206hashset.c)	222

3.1	Sắp xếp theo cách chèn (301insert_s.c)	237
3.2	Sắp xếp theo cách chèn (302inserts2.c)	238
3.3	Sắp xếp theo cách chèn (303insert_b.c)	238
3.4	Thuật toán shell (304shell.c)	240
3.5	Thuật toán shell (305shell2.c)	242
3.6	Phương pháp bong bóng (306bubsort1.c)	244
3.7	Phương pháp bong bóng (307bubsort2.c)	245
3.8	Sắp xếp bằng cách lắc (308shaker.c)	245
3.9	Sắp xếp nhanh (309qsort.c)	250
3.10	Thuật toán thỏ và rùa (310combsort.c)	257
3.11	Sắp xếp lựa chọn trực tiếp (311selsort.c)	259
3.12	Sắp xếp lựa chọn trực tiếp (312selsort2.c)	260
3.13	Sắp xếp kiểu kim tự tháp (313heapsort.c)	265
3.14	Sắp xếp theo tập hợp (314setsort.c)	269
3.15	Sắp xếp theo tập hợp (315setsort2.c)	271
3.16	Sắp xếp theo số đếm (316counts.c)	272
3.17	Sắp xếp theo số đếm (317counts2.c)	274
3.18	Sắp xếp theo bit (318bitsort.c)	278
3.19	Sắp xếp theo bit (319bitsort2.c)	281
3.20	Sắp xếp theo hệ số đếm (320radsort.c)	283
3.21	Sắp xếp theo hóa vị(321permso	288

4.1	Tìm kiếm liên tiếp (401seq-arr.c)	306
4.2	Tìm kiếm liên tiếp (402seq-list.c)	309
4.3	Tìm kiếm liên tiếp với sự sắp xếp lại (403reorder.c)	312
4.4	Tìm kiếm theo bước(404jumpsear.c)	314
4.5	Tìm kiếm nhị phân(405binsear0.c)	318
4.6	Tìm kiếm nhị phân(406binsear1.c)	318
4.7	Tìm kiếm nhị phân(407binsear2.c)	320
4.8	Tìm kiếm nhị phân(408binsear3.c)	320
4.9	Tìm kiếm nhị phân(409binsear4.c)	321
4.10	Tìm kiếm fibonacci(410fibsear.c)	324
4.11	Tìm kiếm nội suy(411interpol.c)	327

5.1	Duyệt theo chiều rộng(501 bfs.c)	348
5.2	Duyệt theo chiều sau(502 dfs.c)	351
5.3	Đường đi ngắn nhất(503 bfsminw.c)	355
5.4	Đường đi ngắn nhất(504 formcycl.c)	358
5.5	Tìm tất cả đường dẫn(505 btdfs.c)	361
5.6	Thuật toán Ford-Bellman (506 belman.c)	366
5.7	Thuật toán floyd (507 floyd.c)	368
5.8	Thuật toán dijkstra (508 dijkstra.c)	375
5.9	Đường dài nhất (509 longpath.c)	384
5.10	Chu trình đơn (510 allcyc.c)	387
5.11	Chu trình Hamilton nhỏ nhất (510 tps.c)	392
5.12	Chu trình Euler (511 eurler.c)	398
5.13	Thuật toán Ford-Fulkerson (512 fordfulk.c)	405

5.14	Định hướng bắc cầu (513trans-or.c)	415
5.15	Kiểm soát công ty (514company.c)	421
5.16	Sắp xếp tó pô (515topsort.c)	425
5.17	Sắp xếp tó pô đầy đủ (516topsortf.c)	427
5.18	Số thành phần liên thông (517strcon1.c)	433
5.19	Các thành phần liên thông mạnh (518strconn.c)	436
5.20	Các điểm phân tách trong đồ thị (519artic.c)	441
5.21	Cây bao phủ nhỏ nhất trong đồ thị (520kruskal.c)	448
5.22	Thuật toán Prim tìm cây bao trùm (521prim.c)	452
5.23	Tập ộc lập cực đại trong đồ thị (522maxindep.c)	457
5.24	Tập trội tối thiểu trong đồ thị (523mindom.c)	461
5.25	Các đỉnh tạo thành tập cơ sở trong đồ thị (524v-base.c)	464
5.26	Tâm và bán kính của đồ thị (525a-center.c)	468
5.27	Tìm p -tâm trong đồ thị trong đồ thị (524526p-center.c)	472

6.1	Biểu thức boolean (601bool.c)	502
6.2	Biểu thức boolean (602boolcut.c)	505
6.3	Tô màu ít nhất của đồ thị (603colorm2.c)	508
6.4	Tìm đường dài nhất (604longpath.c)	512
6.5	Bài toán các bước đi quan ngựa (605knight.c)	516
6.6	Bài toán tám quân Hậu (606queens.c)	521
6.7	Lập thời khóa biểu (607program.c)	527
6.8	Các bản dịch mật mã (608translat.c)	531
6.9	Bài toán ba lô (609bagrec.c)	538
6.10	Trò chơi tic và tắc (610tictac.c)	544

7.1	Tìm phần tử lớn nhất trong (701maximum.c)	561
7.2	Tìm phần tử thứ k (702heap.c)	564
7.3	Tìm phần tử thứ k (703midelem.c)	567
7.4	Tìm theo Hoor (704midel2.c)	569
7.5	Tìm phần tử trội 1(705major1.c)	573
7.6	Tìm phần tử trội cài tiến 2(706major2.c)	574
7.7	Tìm phần tử trội cài tiến 3(707major3.c)	575
7.8	Tìm phần tử trội cài tiến 4(708major4.c)	576
7.9	Tìm phần tử trội cài tiến 5(709major5.c)	576
7.10	Tìm phần tử trội cài tiến 6 (710major6.c)	577
7.11	Tìm phần tử trội cài tiến 7 (711major7.c)	578
7.12	Tìm phần tử trội cài tiến 8 (712major8.c)	581
7.13	Tìm phần tử trội cài tiến 9 (713major9.c)	582
7.14	Tìm phần tử trội cài tiến 10 (714major10.c)	583
7.15	Tìm phần tử trội cài tiến 11 (715major11.c)	584
7.16	Tìm phần tử trội cài tiến 12 (716major12.c)	586
7.17	Tìm phần tử trội cài tiến 13 (717major13.c)	588
7.18	Trộn hai mảng thứ tự làm một (718mergearr.c)	592
7.19	Trộn hai mảng nhị phân (719binmerge.c)	595
7.20	Sắp xếp theo hợp nhất (720mergea.c)	598
7.21	Sửa sắp xếp theo hợp nhất (721mergel1.c)	600
7.22	Sửa sắp xếp theo hợp nhất (722mergel2.c)	602
7.23	Sửa sắp xếp theo hợp nhất (723mergel3.c)	604
7.24	Nâng nhanh lũy thừa (724powerc.c)	608
7.25	Bài toán tháp Hà Nội (725hanoi.c)	621
7.26	Lịch thi đấu giải vô địch(726tourn1.c)	626
7.27	Sửa đổi hàm findSolution() (727tourn2.c)	629
7.28	Sửa đổi hàm findSolution() (728tourn3.c)	630
7.29	Dịch chuyển tuần hoàn của các phần tử mảng (729shift1.c)	632

- 7.30 Dịch chuyển tuần hoàn của các phần tử mảng (730shift2.c)634
- 7.31 Dịch chuyển tuần hoàn của các phần tử mảng (731shift3.c)635