TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP THỰC PHẨM TP.HCM
KHOA CÔNG NGHỆ THÔNG TIN
BỘ MÔN CÔNG NGHỆ PHẦN MỀM

Giáo trình

KỸ THUẬT LẬP TRÌNH NÂNG CAO

(Dành cho hệ Đại học)

TP.HCM, tháng 9 năm 2013

MỤC LỤC

LỜI NÓI ĐẦU

"Algorithm + Data structure = Program"

("Giải thuật + Cấu trúc dữ liệu = Chương trình")

Câu nói nổi tiếng của Niklaus Wirth, một nhà khoa học máy tính nổi tiếng, tác giả của ngôn ngữ lập trình Pascal, đã đặt tên cho một cuốn sách của ông. Đây là một trong những quyển sách nổi tiếng, được làm giáo trình giảng dạy ở các trường đại học lớn. Qua đó chúng ta thấy vai trò quan trọng của giải thuật và kỹ thuật lập trình để xây dựng các giải thuật nhằm tìm đáp án tối ưu nhất cho các bài toán lập trình.

Môn học Kỹ thuật lập trình nâng cao được bố trí sau 2 môn học Kỹ thuật lập trình và Cấu trúc dữ liệu trong chương trình đào tạo cho sinh viên chuyên ngành Công nghệ thông tin. Môn học nhằm giới thiệu cho sinh viên những kiến thức cơ bản, những kỹ thuật chủ yếu của việc phân tích và xây dựng các giải thuật, để tìm ra các cách giải tối ưu nhất có thể cho bài toán. Các kỹ thuật được trình bày ở đây là những kỹ thuật cơ bản, phổ biến và được các nhà khoa học tin học tổng kết và vận dụng trong cài đặt các chương trình. Việc nắm vững các kỹ thuật đó sẽ rất bổ ích cho sinh viên khi phải giải quyết một vấn đề thực tế.

Nội dung giáo trình gồm các phần như sau:

- Chương 1: Tổng quan kỹ thuật lập trình
- Chương 2: Xử lý cấu trúc mảng
- Chương 3: Kỹ thuật đệ qui
- Chương 4: Xử lý chuỗi
- Chương 5: Thiết kế thuật toán

Các vấn đề được trình bày chi tiết với những ví dụ rõ ràng. Cuối mỗi chương có phần bài tập được sắp xếp từ cơ bản đến nâng cao giúp sinh viên nắm vững và kiểm tra kiến thức bằng việc giải các bài tập. Chúng tôi mong rằng các sinh viên tự tìm hiểu trước mỗi vấn đề, kết hợp với bài giảng trên lớp của giảng viên và làm bài tập để việc học môn này đạt hiệu quả.

Trong quá trình giảng dạy và biên soạn giáo trình này, chúng tôi đã nhận được nhiều đóng góp quý báu của các đồng nghiệp ở Bộ môn Công nghệ Phần mềm cũng như các đồng nghiệp trong và ngoài Khoa Công nghệ Thông tin. Chúng tôi xin cảm ơn và hy vọng rằng giáo trình này sẽ giúp cho việc giảng dạy và học môn "Kỹ thuật lập trình nâng cao" đạt hiệu quả tốt hơn. Chúng tôi hy vọng nhận được nhiều ý kiến đóng góp để giáo trình ngày càng hoàn thiện.

Nhóm tác giả

CHƯƠNG 1. TỔNG QUAN KỸ THUẬT LẬP TRÌNH

1.1 Tổng quan về kỹ thuật lập trình

1.1.1 Phong cách lập trình

Một chương trình nguồn được xem là tốt không chỉ được đánh giá thông qua thuật giải đúng và cấu trúc dữ liệu thích hợp, mà còn phụ thuộc vào phong cách và kỹ thuật mã hoá (coding) của người viết chương trình.

Nếu một người lập trình viết một chương trình dù thực hiện đúng yêu cầu đặt ra nhưng mã nguồn quá lộn xộn và phong cách lập trình cẩu thả, thì mã nguồn này sẽ gây khó khăn không chỉ cho những người khác muốn đọc hiểu nó, mà còn cho chính người lập trình khi muốn chỉnh sửa hoặc cải tiến.

Đôi khi người mới lập trình không quan tâm đến vấn đề này do ban đầu chỉ làm việc với chương trình nhỏ. Tuy nhiên, vấn đề phát sinh khi họ phải làm việc với dự án lớn và chương trình lúc này không còn đơn giản vài chục dòng lệnh nữa. Nếu không rèn luyện một phong cách và trang bị một số kỹ thuật lập trình tốt thì người lập trình đối mặt với nhiều khó khăn…

Trong chương đầu tiên xin giới thiệu một số kỹ thuật và phong cách lập trình cơ bản, ít nhiều giúp cho người học viết chương trình được tốt hơn.

1.1.2 Một số kỹ thuật và phong cách lập trình căn bản.

1.1.2.1. Cách đặt tên biến

Thông thường tùy theo ngôn ngữ và môi trường lập trình, người viết chương trình thường chọn cho mình một phong cách nhất quán trong việc đặt tên các định danh. Một số quy tắc cần quan tâm khi đặt tên như sau:

– Tên của định danh phải thể hiện được ý nghĩa: thông thường các biến nguyên như i, j, k dùng làm biến chạy trong vòng lặp; x, y dùng làm biến lưu tọa độ, hoặc dùng làm biến đại diện cho các số bất kỳ… Còn những biến lưu trữ dữ liệu khác thì nên đặt gợi nhớ, nhưng tránh dài dòng: biến đếm số lần dùng "count, dem, so_luong…", biến lưu trọng lượng “weight, trong_luong”, chiều cao “height” ; … Nếu đặt quá ngắn gọn như c cho biến đếm, hay w cho khối lượng thì sau này khi nhìn vào chương trình sẽ rất khó hiểu ý nghĩa của chúng. Ngược lại đặt tên quá dài như "the_first_number, the_second_number,…" để chỉ các số bất kỳ, sẽ làm dư thừa, rườm rà trong chương trình.

– Tên phải xác định được kiểu dữ liệu lưu trữ: phong cách lập trình tốt là khi người đọc nhìn vào một biến nào đó thì xác định ngay được kiểu dữ liệu và tên đối tượng mà biến đó lưu trữ. Cho nên tên biến thường là danh từ (tên đối tượng) kèm theo tiền tố mang ý nghĩa kiểu dữ liệu. Giả sử có biến đếm số lần thì ta có thể đặt iNumber, trong đó i là kiểu của dữ liệu, strContent là kiểu chuỗi, CPoint là lớp Point…Có nhiều cú pháp quy ước đặt tên biến, người lập trình có thể chọn cho mình một quy ước thích hợp. Có thể tham khảo một số quy ước trong phần bên dưới.

– Theo một quy ước cụ thể:

+ Cú pháp Hungary: hình thức chung của cú pháp này là thêm tiền tố chứa kiểu dữ liệu vào tên biến. Bảng 1.1 bên dưới là một số tiền tố quy ước được nhiều lập trình viên sử dụng. Các công ty phần mềm thường có các quy ước về cách đặt tên biến cho

đội ngũ lập trình viên. Tuy nhiên đa số các quy ước này đều dựa trên cú pháp Hungary.

Tiền tố	Kiểu dữ liệu	Ví dụ minh họa
b	bool	bool bEmpty, bChecked ;
c	char	char cInChar, cOutChar ;
str/s	String	string strFirstName, strIn, strOut ;
i/n	integer	int iCount, nNumElement ;
li	long integer	long liPerson, liStars ;
f	float	float fPercent ;
d	double	double dMiles, dFraction ;
if	Input file stream	ifstream ifInFile ;
of	Output file stream	ofstream ofOutFile ;
S	Struct	struct sPoint{…} ;
C	Class	class CStudent,CPerson

+ Đối với những hằng thì tất cả các ký tự đều viết HOA.

Ví dụ 1.1:

```
#define MAXSIZE 100
const float PI = 3.14 ;
```

+ Cách đặt tên cho hàm : hàm bắt đầu với ký tự đầu tiên là ký tự viết thường và các ký tự đầu từ phía sau viết hoa, hoặc các từ cách nhau bằng dấu _ (underscore) và không có tiền tố. Tuy nhiên điều này cũng không bắt buộc tùy theo ngôn ngữ lập trình. Ngoài ra hàm có chức năng thực hiện một nhiệm vụ nào đó, cho nên tên chúng là động từ hoặc cụm động từ, thường bắt đầu bằng các động từ chính: get, set, do, is, make…

Ví dụ 1.2:

```
string setName();
int  countElement ();
void importArr();
```

1.1.2.2. Phong cách viết mã nguồn

– Sử dụng tab để canh lề chương trình : khi soạn thảo mã nguồn nên dùng tab với kích thước là 4 hay 8 khoảng cách để canh lề. Thói quen này giúp cho chương trình được rõ ràng và dễ đọc, dễ quản lý.

Không nên	Nên

void docFile (SV a[], int &n)
{
ifstream in;
char* filename="filein.txt";
in.open (filename);
in>>n;
for(int i=0;i < n;i++)
{
in>>a[i].Masv;
in>>a[i].hoten;
in>>a[i].diem;
}
}
	void docFile (SV a[], int &n)
	{
	ifstream in;
	char* filename = "filein.txt";
	in.open (filename);
	in >> n;
	for (int i=0; i < n; i++)
	{
	in >> a[i].Masv;
	in >> a[i].hoten;
	in >> a[i].diem;
	}
	}

– Sử dụng khoảng trắng : chương trình sẽ dễ nhìn hơn

Không nên	Nên

int iCount =0 ;
for(int i=0;i<n;i++)
{
iCount++;
}
cout<<"Ket qua la:"<<iCount;
	int iCount = 0 ;
	for (int i = 0 ; i < n ; i++)
	{
	iCount ++;
	}
	cout << "Ket qua la:" << iCount;

– Tránh viết nhiều lệnh trên một dòng.

Không nên	Nên

if(a>5){b=a; a++}	if (a > 5)
	{
	b = a;
	a ++;
	}

– Định nghĩa các hằng số.

Một số lập trình có thói quen không định nghĩa những hằng số thường xuyên sử dụng. Dẫn đến những con số khó hiểu xuất hiện trong chương trình, một số tài liệu lập trình gọi những con số này là “magic mumber”.

Không nên	Nên

...
for (int = 0; i < 100; i ++)
a[i] = Rand (100);
...
k = InputNum();
int j = 0;
while (A[j] != k && j < 100)
j ++;
...
	#define MAX_LENGTH 100
	#define MAX_NUM 100
	...
	for (int = 0; i < MAX_LENGTH; i ++)
	a[i] = Rand (MAX_NUM);
	...
	k = InputNum ();
	int j = 0;
	while (a[j] != k && j < MAX_LENGTH)
	j ++;
	...

Trong đoạn chương trình bên trái rất khó phân biệt giá trị 100 ở ba vị trí có mối quan hệ gì với nhau. Tuy nhiên, trong đoạn bên phải ta dễ dàng thấy được ý nghĩa của từng giá trị khi thay bằng định danh. Ngoài ra khi cần thay đổi giá trị của MAX_LENGTH, MAX_NUM thì chỉ cần thay một lần trong phần định nghĩa. Do đó đoạn chương trình bên phải dễ hiểu hơn và dễ thay đổi chỉnh sửa.

– Viết chú thích cho chương trình

Trước và trong khi lập trình cần phải ghi chú thích cho các đoạn mã trong chương trình. Việc chú thích giúp chúng ta hiểu một cách rõ ràng và tương minh hơn, giúp ta dễ đang hiểu khi quay lại chính sửa hoặc cải tiến chương trình. Đặc biệt giúp ta có thể chia sẻ và cùng phát triển chương trình theo nhóm làm việc.

Cụ thể, đối với mỗi hàm và đặc biệt là các hàm quan trọng, phức tạp, chúng ta cần xác định và ghi chú thích về những vấn đề cơ bản sau :

+ Mục đích của hàm là gì ?

+ Biến đầu vào của hàm (tham số) là gì ?

+ Các điều kện ràng buộc của các biến đầu vào (nếu có) ?

+ Kết quả trả về của hàm là gì ?

+ Các ràng buộc của kết quả trả về (nếu có).

+ Ý tưởng giải thuật các thao tác trong hàm.

Ví dụ 1.3 :

Chú thích hợp lý, từng phần làm cho hàm rõ nghĩa, dễ hiểu.

//Hàm tạo danh sách liên kết đôi chứa Phân Số bằng cách đọc dữ liệu từ file txt

```
void  createDList  (DList  & l)
{
     int n;
     ifstream  in;  //biến dùng đọc file
     //tên file chứa dữ liệu đọc vào
     char* filename = "infile.txt";
     in.open  (filename);
     if (in)
     {
          in >> n;
          for (int i = 1;  i<= n;  i++)
          {
               PS x;
               in >> x.ts;
               in >> x.ms;
               if ( x.ms == 0)
                    x.ms  =  rand() % 100 + 1;

     //Tạo node p chứa x và nối p vào sau danh sách l.
               DNode* p = createDNode (x);

               if (l.pHead == NULL)
                    l.pHead = l.pTail = p;
               else
               {
                    p -> pPre = l.pTail;
                    l.pTail -> pNext  = p;
                    l.pTail = p;
               }
          }
     }
in.close();
}
```

Tuy nhiên không phải bất cứ lệnh nào cũng chú thích, việc chú thích tràn lan ngay cả với câu lệnh đơn giản cũng không có ý nghĩa gì. Đôi khi còn làm cho chương trình khó nhìn hơn.

Ví dụ 1.4 :

Không nên chú thích câu lệnh đơn giản này

```
//Nếu nhiệt độ vượt quá mức qui định thì phải cảnh báo
if (nhietDo > nhietDoCB)
     cout<<" Nhiet do vuot muc qui dinh" ;
//i là biến chạy trong vòng lặp for để xác định các chỉ số phần tử mảng a.
for (int i = 0 ; i<n ; i++)
     cout<< a[i] ;
```

– Nên viết biểu thức điều kiện mang tính tự nhiên : biểu thức nên viết dưới dạng khẳng định, việc viết biểu thức dạng phủ định sẽ làm khó hiểu.

Không nên	Nên
`if (!(i < a) || !(i >= b))`	`if ((i >= a) ||  (i < b))`

– Viết các lệnh rõ ràng, tối ưu sự thực thi mã nguồn.

Stt	Không nên	Nên

1	`int i = 0;`
	`while (i < n)`
	`{`
	`  ...`
	`  i++;`
	`}`
		`for(int i = 0; i < n; i++)`
		`{`
		`  ...`
		`}`

2	`i = i + 3 ;` `i +=1 ;`	`i += 3 ;` `i ++ ;`
3	`return (a + b * c) ;`	`return a + b * c ;`

4	`if (a > b)`
	`  return f(a);`
	`else`
	`  return g(b);`
		`return a > b ? f (a) :`
		`g(b) ;`

5	`if (a > b)`
	`  return true ;`
	`else`
	`  return false ;`
		`return a > b ;`

6	`return p.next == NULL ?` `NULL : p.next ;`	`return p.next ;`

7	`if (a > b)`
	`  return`
	`people[current_person].rela`
	`tives.next.`
	`data[x] = f(a);`
	`else`
	`  return`
	`people[current_person].rela`
	`tives.next.`
	`data[x] = f(b);`
		`people[current_person].relat`
		`ives.next.`
		`data[x] = a > b ? f(a) :`
		`f(b) ;`

8	int countNodes (Node *root)
	{
	if (root->left == NULL)
	if (root->right == NULL)
	return 1;
	else
	return 1 + countNodes (root->right);
	else
	if (root->right == NULL)
	return 1 + countNodes (root->left);
	else
	return 1 + countNodes (root->left) + countNodes (root->right);
	}
		int countNodes (Node *root)
		{
		return root == NULL ? 0 : 1 + countNodes (root->left) + countNodes (root->right);
		}

9	F = sqrt (dx * dx + dy * dy) + (sqrt (dx * dx + dy * dy) * sqrt (dx * dx) – sqrt (dy * dy)) ;	A = dx * dx ; B = dy * dy ; C = sqrt (A + B); F = C + (C + sqrt(A) – sqrt (B)) ;
10	for (int i = 0 ; i < strlen(str) ; i++) …	int n = strlen (str) ; for (int i = 0 ; i < n ; i++) …

11	found = FALSE;
	for (int i = 0; i < 10000; i++)
	{
	if (list[i] == -99)
	found = TRUE;
	}
	if (found)
	cout<< "this is a -99";
		found = FALSE;
		for (int i = 0; i < 10000; i++)
		{
		if (list[i] == -99)
		{
		found = TRUE;
		break ;
		}
		}
		if (found)
		cout<< "this is a -99";

12	for(int i = 0; i < n; i ++)
	a[i] = 0;
	for(i = 0; i < n ; i ++)
	b[i] = 0;
		for(i = 0; i < n ; i ++)
		a[i] = b[i] = 0;

– Chia nhỏ chương trình

Trong lập trình nên sử dụng chiến lược " chia để trị", nghĩa là chương trình được chia nhỏ ra thành các chương trình con. Việc chia nhỏ ra thành các chương trình con làm tăng tính modun của chương trình và mang lại cho người lập trình khả năng tái sử

dụng mã code. Các chuyên gia khuyên rằng độ dài mỗi chương trình con không nên vượt quá một trang màn hình để lập trình viên có thể kiểm soát tốt hoạt động của chương trình con đó.

– Hạn chế dùng biến toàn cục.

Xu hướng chung là nên hạn chế sử dụng biến toán cục. Khi nhiều hàm cùng sử dụng một biến toàn cục, việc thay đổi giá trị biến toàn cục của mọt hàm nào đó có thể dẫn đến những thay đổi không mong muốn ở các hàm khác. Biến toàn cục sẽ làm cho các hàm trong chương trình không độc lập với nhau.

1.2 Phân tích đánh giá giải thuật

1.2.1 Sự cần thiết phân tích thuật giải

Trong khi giải một bài toán chúng ta có thể nhiều giải thuật khác nhau, vấn đề là cần phải đánh giá các giải thuật đó để lựa chọn một giải thuật tốt nhất có thể. Thông thường thì ta sẽ căn cứ vào các tiêu chuẩn sau:

- Giải thuật đúng đắn.
- Giải thuật đơn giản.
- Giải thuật thực hiện nhanh.

Với yêu cầu thứ nhất, để kiểm tra tính đúng đắn của giải thuật chúng ta có thể cài đặt giải thuật đó và cho thực hiện trên máy với một số bộ dữ liệu mẫu rồi lấy kết quả thu được so sánh với kết quả cần đạt được. Tuy nhiên cách làm này không chắc chắn vì có thể giải thuật đúng với tất cả các bộ dữ liệu chúng ta đã thử nhưng lại sai với một bộ dữ liệu nào đó mà ta xác định được. Ngoài ra cách này chỉ giúp ta phát hiện giải thuật sai, chứ chưa chứng minh được là nó đúng. Tính đúng đắn của giải thuật cần phải được chứng minh bằng toán học. Điều này không đơn giản với một số bài toán phức tạp.

Khi chúng ta viết một chương trình để sử dụng một vài lần thì yêu cầu thứ 2 là quan trọng nhất. Chúng ta cần một giải thuật dễ viết chương trình để nhanh chóng có được kết qua, thời gian thực hiện chương trình không được đề cao vì dù sao thì chương trình đó cũng chỉ sử dụng một vài lần mà thôi. Tuy nhiên khi một chương trình được sử dụng nhiều lần thì yêu cầu tiết kiệm thời gian thực hiện chương trình lại rất quan trọng đặc biệt đối với những chương trình mà khi thực hiện cần dữ liệu nhập lớn do đó yêu cầu thứ 3sẽ được xem xét một cách kĩ càng. Ta gọi nó là hiệu quả thời gian thực hiện của giải thuật.

1.2.2 Thời gian thực hiện của chương trình

Một phương pháp để xác định hiệu quả thời gian thực hiện của một giải thuật là lập trình nó và đo lường thời gian thực hiện của hoạt động trên một máy tính xác định đối với tập hợp được chọn lọc các dữ liệu vào.

Thời gian thực hiện không chỉ phụ thuộc vào giải thuật mà còn phụ thuộc vào tập các chỉ thị của máy tính, chất lượng của máy tính và kĩ xảo của người lập trình. Sự thi hành cũng có thể điều chỉnh để thực hiện tốt trên tập đặc biệt các dữ liệu vào được chọn. Để vượt qua các trở ngại này, các nhà khoa học máy tính đã chấp nhận

tính phức tạp của thời gian được tiếp cận như một sự đo lường cơ bản sự thực thi của giải thuật. Thuật ngữ tính hiệu quả sẽ đề cập đến sự đo lường này và đặc biệt đối với sự phức tạp thời gian trong trường hợp xấu nhất.#

1.2.2.1 Khái niệm

Thời gian thực hiện một chương trình là một hàm của kích thước dữ liệu vào, ký hiệu T(n) trong đó n là kích thước (độ lớn) của dữ liệu vào.

Ví dụ 1.5:

Chương trình tính tổng của n số có thời gian thực hiện là T(n) = c.n trong đó c là một hằng số. Thời gian thực hiện chương trình là một hàm không âm, tức là $T(n) \geq 0 \; \forall \; n \geq 0$.

1.2.2.2 Thời gian thực hiện trong trường hợp xấu nhất

Thời gian thực hiện một chương trình không chỉ phụ thuộc vào kích thước mà còn phụ thuộc vào tính chất của dữ liệu vào. Nghĩa là dữ liệu vào có cùng kích thước nhưng thời gian thực hiện chương trình có thể khác nhau. Chẳng hạn chương trình sắp xếp dãy số nguyên tăng dần, khi ta cho vào dãy có thứ tự thì thời gian thực hiện khác với khi ta cho vào dãy chưa có thứ tự, hoặc khi ta cho vào một dãy đã có thứ tự tăng thì thời gian thực hiện cũng khác so với khi ta cho vào một dãy đã có thứ tự giảm.

Vì vậy thường ta coi T(n) là thời gian thực hiện chương trình trong trường hợp xấu nhất trên dữ liệu vào có kích thước n, tức là: T(n) là thời gian lớn nhất để thực hiện chương trình đối với mọi dữ liệu vào có cùng kích thước n.

1.2.3 Tỷ suất tăng và độ phức tạp của thuật toán

1.2.3.1 Tỷ suất tăng

Ta nói rằng hàm không âm T(n) có tỷ suất tăng (growth rate) f(n) nếu tồn tại các hằng số C và N_0 sao cho $T(n) \leq Cf(n)$ với mọi $n \geq N_0$.

Ta có thể chứng minh được rằng "*Cho một hàm không âm T(n) bất kỳ, ta luôn tìm được tỷ suất tăng f(n) của nó*".

Ví dụ 1.6:

$$T(n) = \begin{cases} 1 \; với \; n = 0 \\ 4 \; với \; n = 1 \\ (n+1)^2 \; với \; các \; n > 1 \end{cases}$$

Đặt $N_0 = 1$ và C = 4 thì với mọi $n \geq 1$ chúng ta dễ dàng chứng minh được rằng:
$T(n) = (n+1)^2 \leq 4n^2$ với mọi $n \geq 1$, tức là tỷ suất tăng của T(n) là n^2.

Ví dụ 1.7:

Xét tỷ suất tăng của hàm $T(n) = 3n^3 + 2n^2$.

Cho $N_0 = 0$ và $C = 5$ ta dễ dàng chứng minh rằng với mọi $n \geq 0$ thì $3n^3 + 2n^2 \leq 5n^3$. Vậy tỷ xuất tăng của T(n) là n^3.

1.2.3.2 Độ phức tạp của thuật toán

Xét hai giải thuật P1 và P2 với thời gian thực hiện tương ứng là $T1(n) = 100n^2$ (với tỷ suất tăng là n^2) và $T2(n) = 5n^3$ (với tỷ suất tăng là n^3). Giải thuật nào sẽ thực hiện nhanh hơn? Câu trả lời phụ thuộc vào kích thước dữ liệu vào.

Với $n < 20$ thì P2 sẽ nhanh hơn P1 ($T2<T1$), do hệ số của $5n^3$ nhỏ hơn hệ số của $100n^2$ ($5<100$). Nhưng khi $n > 20$ thì ngược lại do số mũ của $100n^2$ nhỏ hơn số mũ của $5n^3$ ($2<3$). Ở đây chúng ta chỉ nên quan tâm đến trường hợp $n>20$ vì khi $n<20$ thì thời gian thực hiện của cả P1 và P2 đều không lớn và sự khác biệt giữa T1 và T2 là không đáng kể.

Như vậy một cách hợp lý là ta xét tỷ suất tăng của hàm thời gian thực hiện chương trình thay vì xét chính bản thân thời gian thực hiện.

Cho một hàm T(n), T(n) gọi là có độ phức tạp f(n) nếu tồn tại các hằng C, N_0 sao cho $T(n) \leq Cf(n)$ với mọi $n \geq N_0$ (tức là T(n) có tỷ suất tăng là f(n)) và kí hiệu T(n) là O(f(n)) (đọc là "ô của f(n)").

Ví dụ 1.8:

$T(n)= 3n^3 + 2n^2$ có tỷ suất tăng là n^3 nên $T(n)= 3n^3 + 2n^2$ là $O(n^3)$.

Lưu ý: $O(C.f(n)) = O(f(n))$ với C là hằng số. Đặc biệt $O(C)=O(1)$.

Nói cách khác độ phức tạp tính toán của giải thuật là một hàm chặn trên của hàm thời gian. Vì C là hằng số trong hàm chặn trên không có ý nghĩa nên ta có thể bỏ qua. Vì vậy hàm thể hiện độ phức tạp có các dạng thường gặp sau: $\log_2 n$, n, $n\log_2 n$, n^2, n^3, 2^n, n!, n^n. Ba hàm cuối cùng ta gọi là dạng hàm mũ, các hàm khác gọi là hàm đa thức. Một giải thuật mà thời gian thực hiện có độ phức tạp là một hàm đa thức thì chấp nhận được tức là có thể cài đặt để thực hiện, còn các giải thuật có độ phức tạp hàm mũ thì phải tìm cách cải tiến giải thuật.

Vì ký hiệu $\log_2 n$ thường có mặt trong độ phức tạp nên ta sẽ dùng **logn** thay thế cho **$\log_2 n$** với mục đích duy nhất là để cho gọn trong cách viết.

Khi nói đến độ phức tạp của giải thuật là ta muốn nói đến hiệu quả của thời gian thực hiện của chương trình nên ta có thể xem việc xác định thời gian thực hiên của chương trình chính là xác định độ phức tạp của giải thuật.

1.2.4 Cách tính độ phức tạp

Cách tính độ phức tạp của một giải thuật bất kỳ là một vấn đề không đơn giản. Tuy nhiên ta có thể tuân theo một số nguyên tắc sau:

1.2.4.1 Qui tắc cộng

Nếu T1(n) và T2(n) là thời gian thực hiện của hai đoạn chương trình P1 và P2; và T1(n)=O(f(n)), T2(n)=O(g(n)) thì thời gian thực hiện của đoạn hai chương trình đó nối tiếp nhau là T(n)=O(max(f(n),g(n)))

Ví dụ 1.9: Lệnh gán x:=15 tốn một hằng thời gian hay O(1), Lệnh đọc dữ liệu READ(x) tốn một hằng thời gian hay O(1).Vậy thời gian thực hiện cả hai lệnh trên nối tiếp nhau là O(max(1,1))=O(1).

1.2.4.2 Qui tắc nhân

Nếu T1(n) và T2(n) là thời gian thực hiện của hai đoạn chương trình P1và P2 và T1(n) = O(f(n)), T2(n) = O(g(n)) thì thời gian thực hiện của đoạn hai đoạn chương trình đó lồng nhau là T(n) = O(f(n).g(n)).

1.2.4.3 Qui tắc tổng quát để phân tích một chương trình.

- Thời gian thực hiện của mỗi lệnh gán, nhập/xuất là O(1).

- Thời gian thực hiện của một chuỗi tuần tự các lệnh được xác định bằng qui tắc cộng. Như vậy thời gian này là thời gian thi hành một lệnh nào đó lâu nhất trong chuỗi lệnh.

- Thời gian thực hiện cấu trúc IF là thời gian kiểm tra điều kiện và thời gian lớn nhất thực hiện khối lệnh sau IF hoặc sau ELSE. Thường thời gian kiểm tra điều kiện là O(1).

- Thời gian thực hiện vòng lặp là tổng (trên tất cả các lần lặp) thời gian thực hiện thân vòng lặp. Nếu thời gian thực hiện thân vòng lặp không đổi thì thời gian thực hiện vòng lặp là tích của số lần lặp với thời gian thực hiện thân vòng lặp.

Ví dụ 1.10:

Tính thời gian thực hiện của giải thuật bubbleSort (Nổi bọt) để sắp xếp mảng 1 chiều a tăng dần.

```
void BubbleSort (int a[], int n)
{
  for(int i = 0; i < n-1;  i++)           (1)
       for(int j = n-1; j > i; j--)       (2)
            if (a[j-1] > a[j] )      (3)
            {
                 int t = a[j-1];          (4)
                 a[j-1] = a[j];      (5)
                 a[j] = t;      (6)
            }
}
```

Ta thấy toàn bộ chương trình chỉ gồm một lệnh lặp (1), lồng trong lệnh (1) là lệnh (2), lồng trong lệnh (2) là lệnh (3) và lồng trong lệnh (3) là 3 lệnh nối tiếp nhau: (4), (5), (6). Chúng ta sẽ tiến hành tính độ phức tạp theo thứ tự từ trong ra.

- Trước hết, cả ba lệnh gán (4), (5), (6) đều tốn O(1) thời gian, việc so sánh a[j-1] > a[j] cũng tốn O(1) thời gian, do đó lệnh (3) tốn O(1) thời gian.

- Vòng lặp (2) thực hiện (n-i) lần, mỗi lần O(1) do đó vòng lặp (2) tốn O((n-i).1) = O(n-i). Vòng lặp (2) lặp có i chạy từ 1 đến n-1 nên thời gian thực hiện của vòng lặp (1) và cũng là độ phức tạp của giải thuật là $T(n) = \sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} = O(n^2)$

Chú ý: Trong trường hợp vòng lặp không xác định được số lần lặp thì chúng ta phải lấy số lần lặp trong trường hợp xấu nhất.

Ví dụ 1.11:

Xét giải thuật tìm kiếm tuyến tính (Linear Search). Hàm tìm kiếm nhận vào một mảng a có n số nguyên và một số nguyên x cần tìm, hàm sẽ trả về giá trị TRUE nếu tồn tại một phần tử a[i] = x, ngược lại hàm trả về FALSE.

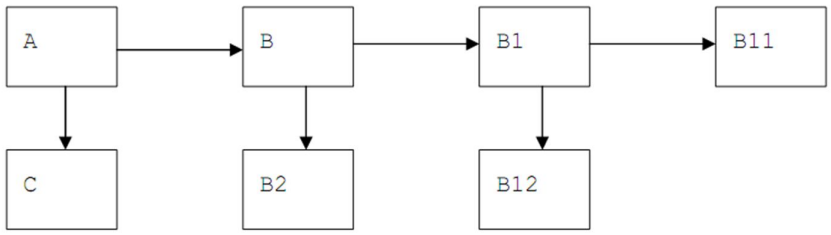
```
bool LinearSearch (int a[], int n, int x)
{
  for (int i = 0; i < n; i++)        (1)
       if(a[i] == x)            (2)
            return true;        (3)

  return false;                 (4)
}
```

Ta thấy các lệnh (1) và (4) nối tiếp nhau, do đó độ phức tạp của hàm chính là độ phức tạp lớn nhất trong 2 lệnh này. Lệnh (4) có độ phức tạp O(1) do đó độ phức tạp của hàm Linear Search chính là độ phức tạp của lệnh (1). Lồng trong lệnh (1) là lệnh (2), lệnh (2) có lệnh lồng là lệnh (3). Lệnh (2) có độ phức tạp là O(1). Trong trường hợp xấu nhất (tất cả các phần tử của mảng a đều khác x) thì vòng lặp (2) thực hiện n lần, vậy ta có T(n) = O(1.n) = O(n).

1.2.4.3 Độ phức tạp của chương trình có gọi chương trình con không đệ qui

Nếu chúng ta có một chương trình với các chương trình con không đệ quy, để tính thời gian thực hiện của chương trình, trước hết chúng ta tính thời gian thực hiện của các chương trình con không gọi các chương trình con khác. Sau đó chúng ta tính thời gian thực hiện của các chương trình con chỉ gọi các chương trình con mà thời gian thực hiện của chúng đã được tính. Chúng ta tiếp tục quá trình đánh giá thời gian thực hiện của mỗi chương trình con sau khi thời gian thực hiện của tất cả các chương trình con mà nó gọi đã được đánh giá. Cuối cùng ta tính thời gian cho chương trình chính.

Giả sử ta có một hệ thống các chương trình gọi nhau theo sơ đồ sau:



Chương trình A gọi hai chương trình con là B và C, chương trình B gọi hai chương trình con là B1 và B2, chương trình B1 gọi hai chương trình con là B11 và B12. Để tính thời gian thực hiện của A, ta tính theo các bước sau:

- Bước 1: Tính thời gian thực hiện của C, B2, B11 và B12. Vì các chương trình con này không gọi chương trình con nào cả.

- Bước 2: Tính thời gian thực hiện của B1. Vì B1 gọi B11 và B12, thời gian thực hiện của B11 và B12 đã được tính ở bước 1.

- Bước 3: Tính thời gian thực hiện của B. Vì B gọi B1 và B2, thời gian thực hiện của B1 đã được tính ở bước 2 và thời gian thực hiện của B2 đã được tính ở bước 1.

- Bước 4: Tính thời gian thực hiện của A. Vì A gọi B và C, thời gian thực hiện của B được tính ở bước 3 và thời gian thực hiện của C đã được tính ở bước 1.

Ví dụ 1.12:

Ta có thể viết lại chương trình sắp xếp bubble theo dạng gọi hàm con swap (hoán đổi giá trị 2 phần tử) như sau:

```
void BubbleSort (int a[], int n)
{
  for(int i = 0; i < n-1;  i++)           (1)
       for(int j = n-1; j > i; j--)       (2)
            if (a[j-1] > a[j] )      (3)
                 Swap (a[j-1], a[j]);     (4)
}

void Swap (int &x, int &y)
{
  int t = x;     (5)
  x = y;         (6)
  y = t;         (7)
}
```

Trong cách viết trên, chương trình Bubble gọi chương trình con Swap, do đó để tính thời gian thực hiện của Bubble, trước hết ta cần tính thời gian thực hiện của Swap. Thời gian thực hiện của Swap là O(1) vì nó chỉ bao gồm 3 lệnh gán (5), (6), (7).

Trong Bubble, lệnh (3) gọi Swap nên chỉ tốn O(1), lệnh (2) thực hiện n-i lần, mỗi lần tốn O(1) nên tốn O(n-i). Lệnh (1) thực hiện n-1 lần nên $T(n) = \sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} = O(n^2)$.

BÀI TẬP CHƯƠNG 1

Bài 1. Xét các định danh sau đây, định danh nào hợp lý hoặc không hợp lý? Vì sao?

1. Định danh cho các biến chạy trong vòng lặp for:

   thefirstelement, thesecondelement, bienchayvongfor, a, b. X, Y, I, j, k .

2. Định danh cho các đối tượng Node, con trỏ trước sau của Node trong danh sách liên kết:

   * the_next_element_inList, *pNext, the_key_of_Node_in_List, key,

3. Định danh biến mang ý nghĩa chỉ nhiệt độ: tempurature, temp, t.

4. Định danh 2 số biến số nguyên bất kỳ

   theFirstArbitraryNumber & theSecondArbitraryNumber; x & y, a & b, iX & iY, iM & iN.

5. Định danh hàm kiểm tra số nguyên x có là số nguyên tố hay không:

   KTNT, SoNguyenTo, kiemTraSoNguyenTo, kiem_Tra_So_Nguyen_To, checkSoNguyenTo, kiemTraSoNT //hàm kiểm tra 1 số có là số nguyên tố.

6. Định danh lớp chứa đối tượng sinh viên: Class_Sinh_Vien, SinhVien, SV, CSV, CSinhVien

Bài 2. Những đoạn mã nguồn sau đây đã viết tối ưu có thể chưa? Các chuẩn về chú thích, định danh… đã hợp lý chưa? Nếu chưa thì sửa như thế nào?

```
//Khai báo các cấu trúc trong chương trình
 typedef struct structtag_Date
{
       int day; int month; int year;
}
Date;

typedef struct tagNode
{
  Date key;
  struct tagNode *pNext, *pPre;
}DNode;

typedef struct tagList { DNode *pHead, *pTail; }DList;
//-----------------------------------------------------
void xuatDList (DList l)
{
  /*   Hàm xuất dữ liệu trong danh sách liên kết đôi
  */
  if(l.pHead==NULL)
  {
            cout<<"Danh sach rong";
            return;
  }

  DNode* p = l.pHead;
```

```
  while (p!=NULL)
  {
       cout<<"   " <<p->key.day <<"/" <<p->key.month <<"/"
<<p->key.year <<"   ";
       p=p->pNext;
  }
}

//-------------------------------------------
/*
   Hàm kiểm tra một Date x có hợp lệ không, Date hợp lệ khi giá trị có trong
lịch dương.
Ví dụ: 31/4/2013, 29/2/2013 là không hợp lệ; 31/5/2011, 29/2/2012 là hợp lệ
Nếu ngày hợp lệ thì trả kết quả true, ngược lại là false.
*/

bool ktraHL (Date x)
{
  //loại bỏ các giá trị không thuộc ngày, tháng, năm
       if(x.day<1  ||  x.day>31  ||  x.month<1  ||x.month>12
|| x.year<1)
            return false;
       else
            if(x.month==2)
            //Năm nhuần tháng 2 có 29 ngày, ngược lại 28
ngày
            {
                  //nam nhuan
                  if(x.year%400==0  ||  (x.year%4==0  &&
x.year%100!=0))
                    if (x.day < 30);
                         return true;
                    else
                         return false;
                else
                    if (x.day < 29)
                         return true;
                    else
                         return false;

            }
            else
                 //thang 4 6 9 11 có 30 ngày
if( x.month ==4 ||x.month==6 ||x.month==9 || x.month ==11)
                    if (x.day<3)
                         return true;
                    else
                         return false;
               else
                    //những tháng có 31 ngày
                    if(x.day<32)
                         return true;
```

```
                    else
                         return false;
}
```

Bài 3. Tính thời gian thực hiện của các đoạn chương trình sau:

1. Tính tổng các phần tử của mảng 1 chiều a gồm n số nguyên

```
int tongM1C (int a[], int n)
        {
       int S = 0;
  for (int i = 0; i<n; i++)
       S += a[i];
  return S;
}
```

2. Tính tích 2 ma trận vuông cấp n: C = A*B

```
void   tich2MaTran  (int  A[20][20],  int  B[20][20],  int
C[20][20], int n)
{
  for(int i=0; i<n; i++)
     for(int j=0; j<n; j++)
     {
       C[i][j] = 0;
            for(int k=0; k<n; k++)
                 C[i][j] += A[i][k] *B[k][j];
     }
}
```

Bài 4. Cho một mảng n số nguyên được sắp thứ tự tăng. Viết hàm tìm một số nguyên trong mảng đó theo phương pháp tìm kiếm nhị phân, nếu tìm thấy thì trả về TRUE, ngược lại trả về FALSE. Sử dụng hai kĩ thuật là đệ quy và vòng lặp. Với mỗi kĩ thuật hãy viết một hàm tìm và tính thời gian thực hiện của hàm đó.

Bài 5. Tính thời gian thực hiện của giải thuật đệ quy giải bài toán Tháp Hà nội (mục 3.5.4) với n tầng?

CHƯƠNG 2. KỸ THUẬT XỬ LÝ MẢNG

2.1 Kỹ thuật xử lý mảng một chiều

Mảng 1 chiều là một dãy các phần tử có cùng tên, cùng kiểu dữ liệu được đặt liên tiếp nhau trong bộ nhớ và các phần tử được truy xuất thông qua chỉ số của mảng.

Mỗi phần tử được xác định bởi một **chỉ số** biểu thị vị trí của phần tử đó trong mảng. Số lượng phần tử trong mảng được gọi là **kích thước** của mảng. Kích thước của mảng là cố định và phải được xác định trước.

Cú pháp khai báo mảng 1 chiều:

<Kiểu dữ liệu><Tên biến kiểu mảng>[<Số phần tử tối đa trong mảng>]

Khai báo mảng 1 chiều a lưu trữ 10 số nguyên

```
int a[10];
```

Khai báo mảng 1 chiều a lưu trữ 50 số thực

```
float a[50];
```

Khai báo mảng 1 chiều a lưu trữ 10 phân số, mỗi phân số gồm 2 thành phần tử số, mẫu số.

```
typedef struct PhanSo
{
    int tu, mau;
}PS;
PS a[10];
```

Khai báo mảng 1 chiều a lưu trữ 10 sinh viên (thông tin 1 sinh viên gồm: mã sinh viên, tên sinh viên, điểm).

```
typedef struct SinhVien
{
    char MaSV[10];
    char TenSV[10];
    int Diem;
}SV;
SV a[10];
```

Mỗi phần tử trong mảng có thể được truy xuất thông qua chỉ số mảng. Phần tử đầu tiên của mảng luôn có chỉ số 0.

Khai báo mảng 1 chiều a lưu trữ 10 số nguyên, và **gán 5** cho **phần tử đầu tiên**.

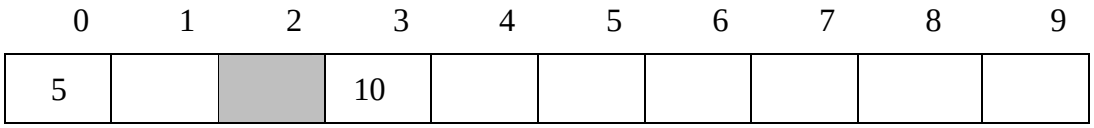
```
int a[10];
    a[0] = 5;
```

Tiếp theo **gán 10** cho **phần tử thứ 3**

```
    a[3] = 10;
```

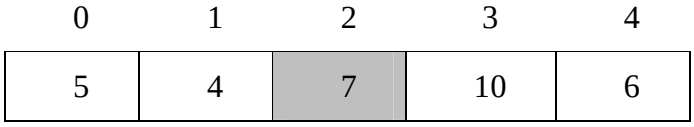
Cho mảng 1 chiều a lưu 10 số nguyên như hình minh họa bên dưới, để **gán 13** cho **ô được tô xám**, ta dùng câu lệnh:

```
a[2] = 13;
```



Cho mảng 1 chiều a lưu 5 số nguyên như hình bên dưới, câu lệnh xuất giá trị số nguyên lưu trong ô nhớ tô màu xám lên màn hình là:

```
printf("%d", a[2]);
```



Cho mảng 1 chiều a lưu trữ 5 số nguyên như hình trên, đoạn lệnh xuất tất cả giá trị lưu trữ trong mảng a lên màn hình là:

```
printf("%d", a[0]);
printf("%d", a[1]);
printf("%d", a[2]);
printf("%d", a[3]);
printf("%d", a[4]);
```

Đoạn lệnh xuất tất cả giá trị lưu trong mảng a được xây dựng bằng cách lặp lại 5 câu lệnh printf để xuất 1 số nguyên ra màn hình, trong đó tên các biến lưu trữ số nguyên có cùng tên a và được phân biệt nhau bởi chỉ số 0, 1, 2, 3, 4. Vì vậy, ta có thể viết lại đoạn chương trình trên bằng cách sử dụng cú pháp lặp for

```
for (int i=0; i<5; i++)
     printf("%d", a[i]);
```

Khi hiển thị các giá trị số nguyên lên màn hình, các số nguyên nằm sát nhau, rất khó phân biệt nhau. Để tạo khoảng cách giữa các số nguyên, ta thêm kí tự định dạng 1 đoạn tab "\t" như sau:

```
for (int i=0; i<5; i++)
     printf("\t %d", a[i]);
```

Tương tự câu lệnh xuất tất cả giá trị số nguyên lên màn hình, đoạn lệnh nhập giá trị từ người dùng vào tất cả phần tử trong mảng là:

```
for (int i=0; i<5; i++)
     scanf("%d",& a[i]);
```

Trước khi nhập giá trị cho phần tử nào, ta cần xuất nhãn để thông báo cho người dùng biết để nhập giá trị vào.

```
for (int i=0; i<5; i++)
{
     printf("nhap a[%d]=", i);
     scanf("%d",& a[i]);
}
```

Việc cố gắng truy xuất một phần tử mảng không tồn tại (ví dụ: a[-1] hoặc a[10]) dẫn tới lỗi thực thi rất nghiêm trọng, được gọi là lỗi 'vượt ngoài biên'.

2.1.1 Thuật toán lặp tổng quát

Để duyệt tất cả các phần tử trong mảng, ta dùng vòng lặp theo cú pháp

Lặp i = 0, 1, 2, 3, … n-1

//Câu lệnh cần lặp

Kết thúc lặp.

Ví dụ 2.1:

Viết hàm nhập mảng 1 chiều số nguyên, với các giá trị người dùng nhập vào.

```
void nhapMang1Chieu(int a[], int &n)
{
     printf("so phan tu mang: n=");
     scanf("%d",&n);
     for (int i=0; i<n; i++)
     {
          printf("a[%d]=",i);
          scanf("%d", &a[i]);
     }
}
```

Ví dụ 2.2:

Viết hàm xuất mảng 1 chiều các phần tử số nguyên

```
void xuatMang1Chieu(int a[], int n)
{
     printf("cac gia tri trong mang la:");
     for (int i=0; i<n; i++)
          printf("%d",a[i]);
}
```

Ví dụ 2.3:

Viết hàm nhập mảng 1 chiều các phần tử kiểu phân số

Phân số là một kiểu dữ liệu tự định nghĩa có 2 thành phần (tử số, mẫu số), nên chương trình không cho phép sử dụng hàm scanf để nhập thông tin trực tiếp vào cho biến p kiểu Phân số (PS). Do đó, để nhập thông tin vào biến p này, bắt buộc phải nhập thông tin lần lượt cho từng thành phần thông qua hàm Nhap_1_Phanso.

```
void nhap_1_PhanSo(PS &p)
{
     printf(" nhap tu so:");
     scanf("%d", &p.tu);
     printf(" nhap mau so:");
     scanf("%d", &p.mau);
}
void nhapMang_PhanSo(PS a[], int &n)
{
     printf("nhap so luong phan so: n=");
     scanf("%d",&n);
```

```
for (int i=0; i<n; i++)
     {
               printf("nhap phan so thu %d: ",i);
               nhap_1_PhanSo(a[i]);
     }
}
```

Ví dụ 2.4:

Viết hàm xuất mảng 1 chiều các phần tử kiểu phân số

Phân số là một kiểu dữ liệu tự định nghĩa không cho phép sử dụng hàm printf xuất thông tin trực tiếp từ biến p kiểu Phân số (PS). Do đó, để xuất thông tin từ biến p này, ta sử dụng hàm Xuat_1_Phanso.

```
void xuat_1_PhanSo(PS p)
{
     printf(" %d/%d", p.tu, p.mau);
}
void xuatMang_PhanSo(PS a[10], int n)
{
     for (int i=0; i<n; i++)
          xuat_1_PhanSo(a[i]);
}
```

Ví dụ 2.5:

Viết hàm nhập mảng 1 chiều các phần tử kiểu sinh viên (thông tin 1 sinh viên gồm: mã sinh viên, tên sinh viên, điểm)

Sinh viên là một kiểu dữ liệu tự định nghĩa có 3 thành phần (mã số, tên, điểm), nên chương trình không cho phép sử dụng hàm scanf để nhập thông tin trực tiếp vào cho biến s kiểu SV, ta sử dụng hàm Nhap_1_SV để nhập thông tin cho 1 sinh viên.

```
void nhap_1_SinhVien(SV &s)
{
     printf(" nhap ma sinh vien:");
     scanf("%s", s.MaSV);
     printf(" nhap tên sinh viên:");
     scanf("%s", s.TenSV);
     printf(" nhap diem:");
     scanf("%d", s.diem);
}
void nhapMang_SinhVien(SV a[10], int &n)
{
     printf("nhap so luong sinh vien: n=");
     scanf("%d",&n);
     for (int i=0; i<n; i++)
     {
          printf("nhap sinh vien thu %d: ",i);
          nhap_1_SinhVien(a[i]);
     }
}
```

Ví dụ 2.6:

Viết hàm xuất mảng 1 chiều các phần tử kiểu sinh viên

Sinh viên là một kiểu dữ liệu tự định nghĩa không cho phép sử dụng hàm printf xuất thông tin trực tiếp từ biến s kiểu Sinh viên (SV). Do đó, để xuất thông tin từ biến s này, ta sử dụng hàm Xuat_1_SinhVien

```
void xuat_1_SinhVien(SV s)
{
  printf("%s\t%s\t%d", s.MaSV, s.TenSV, s.Diem);
}
void xuatMang_SinhVien(SV a[10], int n)
{
  for (int i=0; i<n; i++)
       xuat_1_SinhVien(a[i]);
}
```

2.1.2 Thuật toán tính tổng và tích

2.1.2.1 Tính tổng

Để tính tổng các giá trị trong mảng, ta dùng theo cú pháp

S = 0;

Lặp i = 0, 1, 2, 3, … n-1

S = S + a[i];

Kết thúc lặp.

Ví dụ 2.7:

Viết hàm tính tổng các giá trị trong mảng 1 chiều chứa các giá trị số nguyên

```
int tong(int a[10], int n)
{
  int S=0;
  for(int i=0; i<n; i++)
            S= S+a[i];
  return S;
}
```

Ví dụ 2.8:

Viết hàm tính tổng điểm của tất cả sinh viên trong mảng Sinh viên

```
int tongDiem (SV a[], int n)
{
  int S=0;
  for(int i=0; i<n; i++)
            S = S + a[i].Diem;
  return S;
}
```

Ví dụ 2.9:

Viết hàm tính phân số tổng tất cả phân số trong mảng Phân số

```
PS tongPS (PS a[], int n)
{
  PS p;
```

```
p.tu =0;
p.mau =1;
  for(int i=0; i<n; i++)
  {
       p.tu = p.tu * a[i].mau + p.mau * a[i].tu;
       p.mau = p.mau * a[i].mau;
  }
  return p;
}
```

Để tính tổng các giá trị trong mảng thõa điều kiện đưa ra, ta dùng theo cú pháp

S = 0;
Lặp i = 0, 1, 2, 3, … n-1
    Nếu a[i] thõa điều kiện thì
        S = S + a[i];
Kết thúc lặp.

Ví dụ 2.10:

Viết hàm tính tổng các số có giá trị chẵn trong mảng 1 chiều chứa các số nguyên

```
int tongChan(int a[10], int n)
{
  int S=0;
  for(int i=0; i<n; i++)
       if (i % 2 == 0)     S= S+a[i];
  return S;
}
```

Ví dụ 2.11:

Tính tổng các phân số nhỏ hơn 1 trong mảng chiều chứa n phân số

```
PS tongChanPS(PS a[10], int n)
{
  PS p;
  p.tu = 0;
  p.mau = 1;
  for(int i=0; i<n; i++)
       if ((float)a[i].tu/a[i].mau <1)
       {
            p.tu = p.tu * a[i].mau + p.mau*a[i].tu;
            p.mau = p.mau * a[i].mau;
       }
  return p;
}
```

2.1.2.2 Tính tích

Để tính tích các giá trị trong mảng, ta dùng theo cú pháp

P = 1;

Lặp i = 0, 1, 2, 3, … n-1

P = P * a[i];

Kết thúc lặp.

Ví dụ 2.12:

Viết hàm tính tích các giá trị trong mảng 1 chiều chứa các giá trị số nguyên

```
int tich(int a[10], int n)
{
  int P=1;
  for(int i=0; i<n; i++)
            P = P*a[i];
  return P;
}
```

Ví dụ 2.13:

Viết hàm tính tích các phân số trong mảng Phân số

```
PS tichPS (PS a[], int n)
{
  PS p;
  p.tu = 1;
  p.mau = 1;
  for(int i=0; i<n; i++)
  {
       p.tu = p.tu * a[i].tu;
       p.mau = p.mau * a[i].mau;
  }
  return p;
}
```

Để tính tích các giá trị trong mảng thõa điều kiện đưa ra, ta dùng theo cú pháp

P = 1;

Lặp i = 0, 1, 2, 3, … n-1

Nếu a[i] thõa điều kiện thì

P = P * a[i];

Kết thúc lặp.

Ví dụ 2.14:

Tính tích các số nguyên tố trong mảng 1 chiều chứa các giá trị số nguyên

```
int isPrime(int x)
{
int dem=0;
       if(x<=1)   return 0;
       else
```

```
            for (int i=2;i<=sqrt(x);i++)
            if(x%i==0)     dem++;
        if(dem==0)
            return 1;
    return 0;
}
int tich(int a[10], int n)
{
        int P=1;
        for(int i=0; i<n; i++)
            if (isPrime(a[i])==1)
            P= P*a[i];
        return P;
}
```

Ví dụ 2.15:

Viết hàm kiểm tra xem số nguyên dương n có là số hoàn chỉnh hay không. Một số nguyên dương n là số hoàn chỉnh nếu tổng các ước số của n bằng với tích các ước số của n (không kể n), ví dụ 6 là số hoàn chỉnh vì 6 = 1 + 2 + 3 = 1 * 2 * 3

```
long  tongUocSo(int n)
{
        long S=0; int i;
        for (i = 1; i <n; i++)
            if(n%i==0)
                S = S + i;
        return S;
}
long  tichUocSo(int n)
{
  long P=1; int i;
  for (i = 1; i <n; i++)
        if(n%i==0)
            P = P * i;
  return P;
}
int laSoHoanChinh(int n)
{
  return (tongUocSo(n)==tichUocSo(n));
}
```

2.1.3 Thuật toán đếm

Để đếm các giá trị trong mảng thõa điều kiện đưa ra, ta dùng cú pháp

count = 0;
Lặp i = 1, 2, 3, …, n
    Nếu a[i] thõa điều kiện thì

count = count + 1;

Kết thúc lặp.

Ví dụ 2.16:

Đếm số phần tử dương, phần tử âm, phần tử bằng 0 trong mảng số nguyên a.

```
void   demSo(int a[],int n,int &duong,int &am,int &khong)
{
  int i;
  duong =0; am = 0;   khong = 0;
  for (i = 0; i < n; i++)
  {
       if (a[i]>0)    duong++;
       if (a[i]<0)    ++;
       if(a[i]==0)    khong++;
  }
}
```

2.1.4 Thuật toán tìm phần tử đầu tiên

Để tìm phần tử đầu tiên trong mảng thõa điều kiện đưa ra, ta dùng theo cú pháp

j = -1, i = 0;

Lặp (trong khi (i<n) và (j = -1))

Nếu a[i] thõa điều kiện thì

j = i;

i = i+1;

Cuối lặp.

Ví dụ 2.17:

Tìm phần tử âm đầu tiên trong mảng a mà có tận cùng bằng 6.

```
int     amTanCung6(int a[], int n)
{
  i = 0;
  j = -1;
  while (i < n && j ==-1)
  {
       if(a[i]<0 && -a[i] %10 ==6)
            j = i;
       i++;
  }
  return  j;
}
```

2.1.5 Thuật toán tìm tất cả các phần tử

Để tìm tất cả các phần tử trong mảng thõa điều kiện, ta dùng theo cú pháp

m = 0;

Lặp i = 1, 2, 3, …, n

Nếu (a[i] thõa điều kiện thì)

b[m] = a[i];

m = m + 1;

Cuối nếu.

Cuối lặp.

Ví dụ 2.18:

Tìm tất cả các phần tử chỉ xuất hiện một lần trong mảng số nguyên a, n phần tử.

```
int   dem(int a[], int n, int x)
{
  int i, kq = 0;
  for(i = 0; i < n ; i++)
       if(a[i] == x)
            kq = kq + 1;
  return kq;
}
void  xuatHien01Lan(int a[], int n, int b[], int&m)
{
  m = 0;
  for(i = 0; i < n; i++)
       if(dem(a, n, a[i]) == 1)
       {
            b[m] = a[i]
            m = m + 1;
       }
}
```

2.1.6 Thuật toán tìm min, max

2.1.6.1 Thuật toán tìm min

Bước 1: min := a[0];

Bước 2: Lặp (cho đến khi chọn hết các phần tử của mảng a)

Chọn phần tử a[i], nếu min > a[i] thì gán min = a[i];

Ví dụ 2.19:

Hãy tìm phần tử nhỏ nhất trong mảng 1 chiều các số nguyên a

```
int GTNN (int a[], int n)
{
       i = 0;
       min = a[0];
       while (i < n)
       {
            if(min>a[i])
                      min = a[i];
            i++;
       }
       return min;
  }
```

2.1.6.2 Thuật toán tìm max

Bước 1: max := a[0];

Bước 2: Lặp (cho đến khi chọn hết các phần tử của mảng a)

Chọn phần tử a[i], nếu max < a[i] thì gán max = a[i];

Ví dụ 2.20: Hãy tìm phần tử lớn nhất trong mảng 1 chiều các số nguyên a

```
int GTLN(int a[], int n)
{
	i = 0;
	max = a[0];
	while (i < n)
	{
		if(max < a[i])
			max = a[i];
		i++;
	}
	return max;
}
```

2.1.6.3 Thuật toán tìm min có điều kiện

Bước 1: Chọn một phần tử đầu tiên trong mảng thõa điều kiện x_o, gán min:= x_o

Bước 2: Lặp (cho đến khi chọn hết các phần tử của S)

Chọn phần tử x, nếu x thõa điều kiện và min > x thì gán min := x;

Ví dụ 2.21: Hãy tìm phần tử lẻ nhỏ nhất trong mảng 1 chiều các số nguyên a

```
int timLeDauTien(int a[], int n)
{
  i = 0;
  j = -1;

  while (i < n && j ==-1)
  {
	if(a[i]%2 ==1)
		j = i;
	i++;
  }
  return j;
}
int	MinLe (int a[], int n)
{
  int i, j;
  int min;
  j = TimLeDauTien (a, n);
  if (j ==-1)
	return 0; //không có phần tử lẻ nào
  else
  {
	min = a[j];
	for (i = j+1; i < n; i++)
		if(a[i]%2==1))
			if (min > a[i])
				min = a[i];
  }
```

```
    return min;
  }
```

2.1.6.4 Thuật toán tìm max có điều kiện

Bước 1: Chọn một phần tử đầu tiên trong mảng thõa điều kiện x_o, gán max:= x_o

Bước 2: Lặp (cho đến khi chọn hết các phần tử của S)

Chọn phần tử x, nếu x thõa điều kiện và max < x thì gán max := x;

Ví dụ 2.22:

Hãy tìm phần tử âm lớn nhất có tận cùng là 6 trong mảng a

```
void   maxAmTanCung6 (int a[], int n)
{
  int i, j;
  int max;
  j = AmTanCung6(a, n);
  if (j ==-1) return 0;
  else
  {
       max = a[j];
       for (i = j+1; i < n; i++)
            if(a[i]<0&&(-a[i]%10==6))
                 if (max < a[i])
                      max = a[i];
  }
  return max;
}
```

2.1.7 Thuật toán sắp xếp

2.1.7.1 Sắp xếp mảng

Để sắp xếp mảng 1 chiều tăng dần, có nhiều thuật toán sắp xếp như đổi chỗ trực tiếp (Interchange Sort), chèn trực tiếp (Insertion Sort), chọn trực tiếp (Selection Sort), nổi bọt (Bubble),… Ở đây ta chọn giải thuật đổi chỗ trực tiếp minh họa

Lặp i = 1, 2, …, n -1

Lặp j = i +1, i + 2, …, n

Nếu a[i] > a[j] thì

x = a[i];

a[i] = a[j];

a[j]= x;

Kết thúc nếu.

Kết thúc lặp j.

Kết thúc lặp i.

Ví dụ 2.23:

Viết chương trình sắp xếp các phần tử trong mảng số nguyên tăng dần

```
void sapxeptang(int a[10], int n)
{
       for (int i=0; i<n; i++)
            for (int j=i+1; j<n; j++)
                 if(a[i] > a[j])
                 {
                      int x = a[i];
                      a[i] = a[j];
                      a[j] = x;
                 }
}
```

2.1.7.2 Sắp xếp các phần tử thỏa điều kiện

Lặp i = 1, 2, …, n -1

Nếu (a[i] thõa điều kiện) thì

Lặp j = i +1, i + 2, …, n

Nếu (a[j] thõa điều kiện) thì

Nếu a[i] > a[j] thì

x = a[i];

a[i] = a[j];

a[j]= x;

Cuối lặp j.

Cuối lặp i.

Ví dụ 2.24:

Viết chương trình sắp xếp các phần tử chẵn trong mảng số nguyên tăng dần

```
void sapxepchantang(int a[10], int n)
{
  for (int i=0; i<n; i++)
       if (a[i] % 2 ==0 )
            for (int j=i+1; j<n; j++)
                 if(a[j] %2==0)
                      if(a[i] > a[j])
                      {
                           int x = a[i];
                           a[i] = a[j];
                           a[j] = x;
                      }
}
```

2.2 Kỹ thuât xử lý mảng hai chiều

2.2.1 Mảng hai chiều (ma trận)

Mảng có thể có hơn một chiều, nghĩa là, hai, ba, hoặc cao hơn. Việc tổ chức mảng nhiều chiều trong bộ nhớ cũng tương tự như mảng một chiều (bao gồm một chuỗi liên tiếp các phần tử). Xử lý mảng nhiều chiều thì tương tự như là mảng một chiều nhưng phải xử lý các vòng lặp lồng nhau thay vì vòng lặp đơn.

Để khai báo kiểu dữ liệu mảng 2 chiều ta dùng cú pháp:

<Kiểu dữ liệu> <Tên mảng> [<Số dòng tối đa>][<Số cột tối đa>]

Khai báo mảng 2 chiều kiểu số nguyên gồm 10 dòng, 10 cột

int A[10][10];

Khai báo mảng 2 chiều kiểu số thực gồm 10 dòng, 10 cột

float A[10][10];

Để truy xuất các phần tử của ma trận ta dựa vào chỉ số dòng, chỉ số cột của ma trận

printf("%d", a[2][1]); //xuất ra màn hình giá trị dòng thứ 2, cột thứ 1

Cú pháp duyệt từng dòng từ trên xuống dưới.

```
for(i = 0; i< so_dong; i++)
   for(j = 0; j < so_cot; j++)
   {     //Xử lý a[i][j];
   }
```

Ví dụ 2.25:

Viết chương trình nhập, xuất mảng 2 chiều các phần tử số nguyên

```
void nhapMang2C(int a[][MAX], int &m, int &n)
{
     printf("so dong: m=");        scanf("%d",&m);
     printf("so cot: n=");         scanf("%d",&n);
     for (int i=0; i<m; i++)
          for(int j=0;j<n; j++)
          {
               printf("a[%d][%d]=",i,j);
               scanf("%d", &a[i][j]);
          }
}
void xuatMang2C(int a[][MAX], int m, int n)
{
     for (int i=0; i<m; i++)
     {
          for(int j=0; j<n; j++)
               printf("%d",a[i][j]);
          printf("\n");
     }

}
```

Ví dụ 2.26:

Viết hàm tìm phần tử nhỏ nhất trong ma trận a có m dòng và n cột

```
int GTNN (int  a[][MAX], int m, int n )
```

```
{
  int min = a[0][0];
  for ( int i = 0 ; i < m ; i ++ )
       for (int j = 0 ; j < n ; j ++)
            if ( a[i][j] < min )
                 min = a[i][j];
  return min;
}
```

Duyệt từng cột từ trái qua phải

```
for(j = 0; j < so_cot; j++)
   for(i = 0; i< so_dong; i++)
   {
        //Xử lý a[i][j];
   }
```

Ví dụ 2.27:

Viết chương trình nhập, xuất mảng 2 chiều các phần tử số thực

```
void nhapMang2C (float a[][MAX], int &m, int &n)
{
  printf("so dong: m=");
  scanf("%d",&m);
  printf("so cot: n=");
  scanf("%d",&n);
  for(int j=0;j<n; j++)
       for(int i=0;i<m; i++)
       {
            printf("a[%d][%d]=",i,j);
            scanf("%d", &a[i][j]);
       }
}
void xuatMang2C(int a[][MAX], int m, int n)
{
  for(int j=0; j<n; j++)
  {
       for (int i=0; i<m; i++)
            printf("%d",a[i][j]);
       printf("\n");
  }
}
```

2.2.2 Thuật toán cơ bản trên mảng hai chiều

2.2.2.1 Thuật toán tổng và tích

Để tính tổng có các giá trị trong mảng 2 chiều, ta sử dụng cú pháp:

```
S=0;
for(i = 0; i< so_dong; i++)
     for(j = 0; j < so_cot; j++)
          S=S+ a[i][j];
```

Ví dụ 2.28:

Viết hàm tính tổng các phần tử trong ma trận có m dòng và n cột

```
long tong (int a[][MAX], int m, int n)
{
  long tong = 0;
  for ( int i = 0; i < m; i ++ )
     for ( int j = 0; j < n; j ++)
       tong + = a[i][j];
  return tong;
}
```

Để tính tích có các giá trị trong mảng 2 chiều, ta sử dụng cú pháp:

```
P=1;
for(i = 0; i< so_dong; i++)
     for(j = 0; j < so_cot; j++)
          P=P * a[i][j];
```

Để tính tổng có điều kiện tất cả các giá trị trong mảng 2 chiều, ta sử dụng cú pháp:

```
S=0;
for(i = 0; i< so_dong; i++)
     for(j = 0; j < so_cot; j++)
          //Nếu a[i][j] thoa điều kiện
               S=S + a[i][j];
```

Để tính tích có điều kiện tất cả các giá trị trong mảng 2 chiều, ta sử dụng cú pháp:

```
P=1;
for(i = 0; i< so_dong; i++)
     for(j = 0; j < so_cot; j++)
          //Nếu a[i][j] thoa điều kiện
               P=P * a[i][j];
```

Để tính tổng các phần tử trên 1 dòng thu k, ta dùng theo cú pháp:

```
S=0;
for (i = 0; i < so_cot; i++)
     S= S + a[k][i];
```

Để tính tích các phần tử trên 1 dòng thu k, ta dùng theo cú pháp:

```
P=1;
for (i = 0; i < so_cot; i++)
     P = P * a[k][i];
```

Để tính tổng có điều kiện các phần tử trên 1 dòng thu k, ta dùng theo cú pháp:

```
S=0;
for (i = 0; i < so_cot; i++)
     Nếu a[k][i] thõa điều kiện
          S= S + a[k][i];
```

Để tính tích có điều kiện các phần tử trên 1 dòng thu k, ta dùng theo cú pháp:

```
P=1;
for (i = 0; i < so_cot; i++)
   Nếu a[k][i] thõa điều kiện
```

P= P * a[k][i];

Ví dụ 2.29:

Viết hàm tính tổng các giá trị chẵn trên dòng thứ k

```
int TongChanDongThuK(int a[][MAX], int n, int m, int k)
{
  S=0;
  for (i = 0; i < m; i++)
       if (a[k][i] % 2 ==0)
            S= S + a[k][i];
}
```

2.2.2.2 Thuật toán đếm

Để đếm số lượng các phần tử trong mảng thõa điều kiện, ta dùng cú pháp

Count = 0;

For(int i=0; i<so_dong; i++)

For(int j=0; j<so_cot; j++)

//Nếu a[i][j] thõa điều kiên

Count ++;

Ví dụ 2.30:

Viết hàm đếm số lượng phần tử chẵn trong ma trận có mxn số nguyên

```
int demChan(int a[][MAX], int m, int n)
{
     int Count=0;
     for (int i=0; i<m; i++)
          for( int j =0; j<n; j++)
               if(a[i][j] %2 == 0)
                    Count++ ;
     return Count;
}
```

Ví dụ 2.31:

Viết chương trình đếm số lượng các phần tử dương nằm trên dòng thứ k của ma trận có mxn số nguyên

```
int DemDuongDongk(int a[][MAX], int m, int n, int k)
{
  int Count=0;
  for( int j =0; j<n; j++)
       if(a[k][j] > 0)
                 Count++ ;
       return Count;
}
```

Ví dụ 2.32:

Viết chương trình tìm dòng số lượng các phần tử dương nhiều nhất trong ma trận có mxn số nguyên (nếu có nhiều hơn 2 dòng có số lượng số dương nhiều nhất, trả về kết quả dòng đầu tiên).

```
int DemDuongk(int a[][MAX], int m, int n, int k)
{
  int Count=0;
  for( int j =0; j<n; j++)
       if(a[k][j] > 0)
                 Count++ ;
       return Count;
}

int DongSoDuongMax(int a[][50], int m, int n)
{
  int vtmax = 0;
  for(int i=0; i<m; i++)
     if(DemDuongk(a,m,n,i) > DemDuongk(a,m,n,vtmax))
         vtmax = i;
  return vtmax;
}
```

2.2.2.3 Thuật toán tìm giá trị lớn nhất, giá trị nhỏ nhất

Thuật toán tìm giá trị nhỏ nhất

Bước 1: min := a[0][0];

Bước 2: Lặp (cho đến khi chọn hết các phần tử của mảng a)

Chọn phần tử a[i][j], nếu min > a[i][j] thì gán min = a[i][j];

Ví dụ 2.33:

Hãy tìm phần tử nhỏ nhất trong mảng số nguyên a kích thước mxn

```
int GTNN(int a[][Max], int m, int n)
{
  min = a[0][0];
  for (int i=0; i<m; i++)
       for(int j=0; j<n; j++)
            if(min>a[i][j])
                 min = a[i][j];
  return min;
}
```

Thuật toán tìm giá trị nhỏ nhất có điều kiện

Bước 1: Chọn một phần tử đầu tiên trong mảng thõa điều kiện x_o, gán min:= x_o

Bước 2: Lặp (cho đến khi chọn hết các phần tử của S)

Chọn phần tử x, nếu x thõa điều kiện và min > x thì gán min := x;

Ví dụ 2.34:

Hãy tìm phần tử lẻ nhỏ nhất trong mảng số nguyên a kích thước mxn

```
int TimLeDauTien(int a[][Max], int m, int n)
{
  for(int i=0; i<m; i++)
  for(int j=0; j<n; j++)
       if(a[i][j] %2 ==1)
            return a[i][j];
  return 0;
}
int    MinLe (int a[][Max], int m, int n)
{
  int min;
  min = TimLeDauTien (a, m,n);
  if (min ==0) return 0; //không có phần tử lẻ nào
  for(int i=0; i<m; i++)
       for (j = 0; j < n; j++)
            if(a[i][j]%2==1))
                 if (min > a[i][j])
                      min = a[i][j];
  return min;
}
```

Thuật toán tìm giá trị lớn nhất

Bước 1: max := a[0][0];

Bước 2: Lặp (cho đến khi chọn hết các phần tử của mảng a)

Chọn phần tử a[i][j], nếu max < a[i][j] thì gán max = a[i][j];

Ví dụ 2.35:

Hãy tìm phần tử lớn nhất trong mảng số nguyên a kích thước mxn

```
int GTLN(int a[][50], int m, int n)
{
  max = a[0][0];
  for (int i=0; i<m; i++)
       for(int j=0; j<n; j++)
            if(max<a[i][j])
                 max = a[i][j];
  return max;
}
```

Thuật toán tìm giá trị nhỏ nhất trên dòng i

Cho mảng 2 chiều các số nguyên có kích thước 3x4 như sau

4	7	3	5
2	1	6	9
8	5	9	4

Giá trị nhỏ nhất trên dòng 0 là 3

Giá trị nhỏ nhất trên dòng 1 là 1

Giá trị nhỏ nhất trên dòng 2 là 4

Trong ma trận, mỗi dòng sẽ có 1 giá trị nhỏ nhất khác nhau. Để in giá trị nhỏ nhất trên tất cả các dòng của ma trận, ta dựa vào cách duyệt ma trận tuần tự theo từng dòng của ma trận bắt đầu từ dòng thứ i=0 đến dòng thứ i=m.

```
void printGTNN_Dong(int a[][50], int m, int n)
{
  for(int i=0; i<m; i++)
  {
       int min = a[i][0];
       for(int j=0; j<n; j++)
            if(min > a[i][j])
                 min = a[i][j];
       printf(" Gia tri nho nhat tren dong %d la:
%d",i,min);
  }
}
```

Trong thực tế, đôi lúc chúng ta chỉ quan tâm đến một giá trị nhỏ nhất trên một dòng i cụ thể, mà giá trị i này có thể thay đổi tùy theo yêu cầu người đang sử dụng hàm. Thuật toán sau đây giúp ta tìm giá trị nhỏ nhất trên một dòng i, với i là giá trị người dùng chỉ định.

```
int GTNN_DongI(int a[][50], int m, int n, int i)
{
  int min = a[i][0];
  for(int j=0; j<n; j++)
       if(min > a[i][j])
            min = a[i][j];
  return min;
}
```

Thuật toán tìm giá trị lớn nhất trên cột

Ngược lại thuật toán tìm giá trị nhỏ nhất trên tất cả các dòng trong ma trận, thuật toán tìm giá trị lớn nhất trên tất cả cột dựa vào cách duyệt lần lượt theo từng cột của ma trận bắt đầu từ cột j=0 đến cột thứ j=n như sau

```
void printGTLN_Cot(int a[][50], int m, int n)
{
  for(int j=0; j<n; j++)
  {
       int max = a[0][j];
       for(int i=0; i<m; i++)
            if(max < a[i][j])
                 max = a[i][j];
       printf(" Gia tri lon nhat tren cot %d la:
%d",j,max);
  }
}
```

Thuật toán tìm giá trị lớn nhất trên 1 cột j, trong đó j là giá trị chỉ định từ người dùng

```
int GTLN_CotJ(int a[][50], int m, int n, int j)
{
```

```
    int max = a[0][j];
    for(int i=0; i<n; i++)
        if(max < a[i][j])
            max = a[i][j];
    return max;
}
```

2.2.3 Ma trận vuông

Khi các phần tử trong mảng 2 chiều có số dòng bằng số cột (m=n) ta gọi ma trận đó là ma trận vuông.

Duyệt các phần tử trên đường chéo chính trong mảng 2 chiều vuông

```
for (i = 0; i < n; i++)
  {
        // Xử lý phần tử a[i][i];
  }
```

Ví dụ 2.36:

Viết hàm đếm số phần tử âm trên đường chéo chính

```
int demPTAm (int a[10][10], int n)
{
        int dem = 0;
        for( int i = 0; i<n; i++)
                if(a[i][i] <0)
                        dem ++;
        return dem;
}
```

Để tính tổng có điều kiện tất cả các giá trị trên đường chéo chính trong mảng 2 chiều vuông, ta sử dụng cú pháp:

```
S=0;
for(i = 0; i< so_dong; i++)
        //Nếu a[i][i] thoa điều kiện
                        S=S + a[i][i];
```

Duyệt các phần tử trên đường chéo phụ trong mảng 2 chiều vuông

```
for (i = 0; i < n; i++)
{
    // Xử lý phần tử a[n-1-i][i];
}
```

Ví dụ 2.37:

Viết hàm tính tổng số chẵn trên đường chéo phụ

```
int tongChan (int a[10][10], int n)
{
	int S = 0;
	for( int i = 0; i<n; i++)
		if(a[n-1-i][i] % 2 == 0)
			S = S + a[n-1-i][i];
	return S;
}
```

Để tính tích có điều kiện tất cả các giá trị trên đường chéo phụ trong mảng 2 chiều vuông, ta sử dụng cú pháp:

```
P=1;
for (i = 0; i < so_dong; i++)
	//Nếu a[n-1-i][i] thoa điều kiện
		P=P * a[n-1-i][i];
```

Ví dụ 2.38:

Viết chương trình tính tích các số âm trên đường chéo phụ của mảng 2 chiều vuông có kích thước n.

```
int tichAmDuongCheoPhu(int a[][MAX], int n)
{
	int P=1;
	for (int i=0; i<n; i++)
		if(a[n-1-i][i] < 0)
			P = P*a[n-1-i][i];
		return P;
}
```

Duyệt các phần tử trên dòng thứ k (0 <=k < so_dong)

```
for (j = 0; j < so_cot; j++)
{
  // Xử lý phần tử a[k][j];
}
```

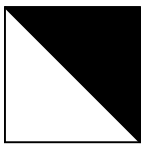
Ví dụ 2.39:

Viết hàm tính tổng số chẵn trên dòng thứ k

```
int tongChan (int a[10][10], int n, int k)
{
	int S = 0;
	for( int j = 0; j<n; j++)
		if(a[k][j] % 2 == 0)
```

```
            S = S + a[n-1-i][i];
        return S;
    }
```

Duyệt các phần tử trong nửa mảng vuông thuộc tam giác trên



Mảng a[i][j]
Điều kiện i ≤j

Cú pháp duyệt nửa mảng

```
for(i = 0; i< so_dong; i++)
   for(j = 0; j < so_cot; j++)
   {
        if (i<j)
        //Xử lý a[i][j];
   }
```

Ví dụ 2.40:

Đếm số phần tử chẵn trong tam giác trên của ma trận vuôngchứa n số nguyên

```
int demChanTGTren (int a[10][10], int n)
{
     int dem =0;
     for(int i=0;i<n; i++)
          for(int j=0;j<n; j++)
          {
               if(i <= j)
                    if(a[i][j] % 2 == 0)
                         dem ++;
          }
     return dem;
}
```

Duyệt các phần tử trong nửa mảng vuông thuộc tam giác dưới

Mảng a[i][j]
Điều kiện i ≥j

Ví dụ 2.41:

Đếm số phần tử âm trong tam giác dưới của ma trận vuông chứa n số thực

```
int demAmTGDuoi (float a[10][10], int n)
```

```
{
  int dem =0;
  for(int i=0;i<n; i++)
       for(int j=0;j<n; j++)
            if(i >= j)
                 if(a[i][j] < 0)
                      dem ++;
  return dem;
}
```

Duyệt các phần tử trong nửa mảng vuông thuộc tam giác trái

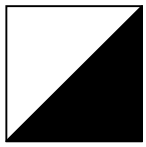
Mảng a[i][j]
Điều kiện i + j ≤ n-1

Ví dụ 2.42:

Đếm số phần tử âm trong tam giác trái của ma trận vuông chứa n số thực

```
int demAmTGTrai (float a[10][10], int n)
{
  int dem =0;
  for(int i=0;i<n; i++)
       for(int j=0;j<n; j++)
            if(i+j <= n-1)
                 if(a[i][j] < 0)
                      dem ++;
  return dem;
}
```

Duyệt các phần tử trong nửa mảng vuông thuộc tam giác phải



Mảng a[i][j]
Điều kiện i + j ≥ n-1

Ví dụ 2.43:

Đếm số phần tử chẵn trong tam giác phải của ma trận vuông chứa n số nguyên

```
int demChanTGPhai (int a[10][10], int n)
{
  int dem =0;

  for(int i=0;i<n; i++)
       for(int j=0;j<n; j++)
       {
            if(i+j >= n-1)
                 if(a[i][j] % 2 == 0)
                      dem ++;
```

```
        }

    return dem;
}
```

2.2.4 Một số bài toán đặc biệt

2.2.4.1 Phần tử yên ngựa

Trong một ma trận 2 chiều có thước m x n, một phần tử a[i][j] được gọi là điểm yên ngựa của ma trận nếu như nó là phần tử nhỏ nhất trên dòng i và lớn nhất trên cột j của ma trận hoặc nó là phần tử lớn nhất trong dòng i và nhỏ nhất trên cột j của ma trận.

Ví dụ, trong ma trận 3x3 sau đây, số 6 là điểm yên ngựa của ma trận

6	7	8
5	4	9
3	2	4

Giải thuật

```
for (i=0;i<n;i++)
{
     for (j=0;j<m;j++)
     {
          if ((a[i][j]==GTNN_DongI(a,m,n,i) && a[i][j] ==
     GTLN_CotJ(a,m,n,j)) || (a[i][j]==GTLN_DongI(a,m,n,i)
     && a[i][j] == GTNN_CotJ(a,m,n,j)))
                    cout << a[i][j];
     }
}
```

2.2.4.2 Phần tử hòang hậu

Trong ma trận 2 chiều có kích thước mxn, một phần tử a[i][j] được gọi là phần tử hoàng hậu nếu như nó lớn nhất trên dòng, lớn nhất trên cột và lớn nhất trên hai đường chéo chứa nó.

Xét ma trận sau:

6	7	2
5	1	3
3	2	4

Phần tử 7 lớn nhất trên dòng 0, lớn nhất trên cột 1, và lớn nhất trên đường chéo 5,7 và 7,3. Phần tử 7 là phần tử hoàng hậu trên ma trận.

Phần tử 5 lớn nhất trên dòng 1 nhưng không lớn nhất trên cột 0. Phần tử 5 không là phần tử hoàng hậu.

Phần tử 4 không là phẩn tử hoàng hậu vì không là phần tử lớn nhất trên đường chéo 6,1,4.

Giải thuật

```
int XetDuongCheo(int a[][100],int dong,int cot, int i, int j)
     {
          int k,h; //k : dong, h : cot
```

```
          for (k=i,h=j; k>=0 && h<cot; k--, h++)
               if (a[k][h] > a[i][j])
                    return 0;
          for (k=i+1, h=j-1; k<dong && h>=0; k++, h--)
               if (a[k][h] > a[i][j])
                    return 0;
          for (k=i-1, h =j-1; k>=0 && h>=0; k--,h--)
               if (a[k][h] > a[i][j])
                    return 0;
          for (k=i+1, h=j+1; k<dong && h<cot; k++, h++)
               if (a[k][h] > a[i][j])
                    return 0;
          return 1;
     }

int XetCotDong(int a[][100],int dong,int cot, int i, int j)
     {
          for (int k=0; k<dong; k++ )
               if (a[k][j]>a[i][j])
                    return 0;
          for (k=0; k<cot; k++ )
               if (a[i][k]>a[i][j])
                    return 0;
          return 1;
     }
int XetTongQuat(int a[][MAX],int dong,int cot, int i, int j)
     {
          if (XetCotDong(a,dong,cot,i,j)==1)
               if (XetDuongCheo(a,dong,cot,i,j)==1)
                    return 1;
          return 0;
     }
```

2.2.4.3 Phần tử cực đại

Một phần tử được gọi là cực đại khi nó lớn hơn tất cả các phần tử lân cận của nó.

```
int ktcucdai(float a[][MAX], int m, int n, int i, int j)
{
    int flag=1;
    for(int di=-1;di<=1;di++)
            for(int dj=-1;dj<=1;dj++)
                    if(i+di>=0 && j+dj>=0 && i+di<m &&
j+dj<n && !(di==0 && dj==0) && a[d][c]<=a[d+di][c+dj])
                    flag =0;
    return flag;
}
```

BÀI TẬP CHƯƠNG 2

Bài 1. Cho mảng 1 chiều a có n phần tử, hãy thực hiện các chức năng sau:

1. Viết hàm liệt kê các giá trị chẵn trong mảng 1 chiều các số nguyên

2. Viết hàm liệt kê các vị trí mà giá trị tại đó là giá trị âm trong mảng 1 chiều các số nguyên.

3. Viết hàm tìm giá trị lớn nhất trong mảng 1 chiều các số thực.

4. Viết hàm tìm giá trị dương đầu tiên trong mảng 1 chiều các số thực. Nếu mảng không có giá trị dương thì trả về giá trị -1.

5. Viết hàm tìm số chẵn cuối cùng trong mảng 1 chiều các số nguyên. Nếu mảng không có giá trị chẵn thì trả về giá trị -1.

6. Viết hàm tìm 1 vị trí giá trị nhỏ nhất đầu tiên trong mảng 1 chiều các số thực.

7. Viết hàm tìm vị trí của giá trị chẵn đầu tiên trong mảng một chiều các số nguyên. Nếu mảng không có giá trị chẵn thì trả về giá trị -1.

8. Viết hàm tìm vị trí số hoàn thiện cuối cùng trong mảng 1 chiều các số nguyên. Nếu mảng không có số hoàn thiện thì trả về giá trị -1.

9. Viết hàm tìm giá trị dương nhỏ nhất trong mảng các số thực. Nếu mảng không có giá trị dương thì trả về giá trị -1.

10. Viết hàm tìm số nguyên tố đầu tiên trong mảng 1 chiều các số nguyên.

11. Viết hàm tìm số hoàn thiện đầu tiên trong 1 chiều các số nguyên.

12. Viết hàm tìm tìm giá trị âm lớn nhất trong mảng các số thực. Nếu mảng không có giá trị âm thì trả về giá trị 0.

13. Viết hàm tìm số nguyên tố lớn nhất trong mảng 1 chiều. Nếu mảng không có số nguyên tố thì trả về giá trị 0.

14. Viết hàm tìm vị trí số hòan thiện nhỏ nhất trong mảng 1 chiều các số nguyên. Nếu mảng không có trả về 0.

15. Viết hàm tìm giá trị chẵn nhỏ nhất trong mảng 1 chiều các số nguyên. Nếu mảng không có giá trị chẵn trả về -1, nếu mảng có nhiều hơn 1 giá trị chẵn nhỏ nhất thì trả về vị trí phần tử chẵn nhỏ nhất cuối cùng.

16. Viết hàm tìm giá trị trong mảng 1 chiều các số thực xa giá trị x nhất

17. Viết hàm tìm 1 vị trí trong mảng 1 chiều các số thực mà giá trị tại vị trí đó là giá trị gần giá trị x nhất

18. Viết hàm tìm đoạn [a,b] sao cho đoạn này chứa tất cả các giá trị trong mảng 1 chiều các chứa các số thực.

19. Viết hàm tìm giá trị x sao cho đoạn [-x,x] chứa tất cả các giá trị trong mảng 1 chiều các số nguyên.

20. Viết hàm tìm giá trị đầu tiên của mảng 1 chiều số nguyên nằm trong khỏang (x,y) cho trước. Nếu ko có trả về giá trị x

21. Viết hàm tìm 1 vị trí trong mảng 1 chiều số nguyên thỏa 2 điều kiện: có giá trị lân cận và giá trị tại đó bằng tích 2 giá trị lân cận. Nếu ko có trả về -1.

22. Cho mảng 1 chiều các số nguyên. Viết hàm tìm giá trị toàn là chữ số lẻ và lớn nhất trong những số thỏa điều kiện. Nếu mảng không có giá trị thõa điều kiện thì trả về 0.

23. Viết hàm tìm số chẵn lớn nhất nhỏ hơn mọi giá trị lẻ có trong mảng 1 chiều số nguyên.

24. Viết hàm liệt kê các số thỏa điều kiện lớn hơn trị tuyệt đối của số đứng liền sau nó trong mảng 1 chiều số nguyên.

25. Cho mảng 1 chiều số nguyên. Viết hàm liệt kê các giá trị thỏa điều kiện nhỏ hơn trị tuyệt đối của giá trị đứng liền sau và lớn hơn trị tuyệt đối số đứng liền trước nó.

Bài 2. Cho ma trận số nguyên a có m dòng, n cột, hãy thực hiện các chức năng sau:

1. Viết hàm tính tổng các phần tử có chữ số đầu là chữ số lẻ

2. Viết hàm đếm các ptử có chữ số hàng chục là 5

3. Viết hàm liệt kê các số hoàn thiện trong ma trận

4. Viết hàm tính tổng các phần tử lớn hơn trị tuyệt đối của phần tử liền sau nó

5. Viết hàm tính tổng các phần tử cực trị cảu ma trận

6. Viết hàm tính tổng giá trị trên 1 dòng ma trận

7. Viết hàm tính tổng giá trị dương trên 1 dòng ma trận

8. Viết hàm tìm tổng các giá trị nằm trên biên của ma trận

9. Viết hàm đếm tần suất xuất hiện của 1 giá trị x trong ma trận

10. Viết hàm đếm số lượng phần tử cực đại trong ma trận.

11. Viết hàm đếm số lượng phần tử cực trị ma trận.

12. Viết hàm đếm số lượng giá trị "hoàng hậu" trên ma trận.

13. Viết hàm đếm số lượng "yên ngựa".

14. Viết hàm kiểm tra ma trận có toàn dương hay không.

15. Viết hàm kiểm tra 1 hàng ma trận có tăng dần hay không

16. Viết hàm kiểm tra 1 cột ma trận có giảm dần không

17. Viết hàm kiểm tra các gía trị trong ma trận có giảm dần theo cột và dòng

18. Viết hàm liệt kê chỉ số các dòng chứa toàn giá trị chẵn

19. Viết hàm liệt kê các dòng chứa giá trị giảm dần

20. Viết hàm tìm giá trị xuất hiện nhiều nhất trong ma trận

21. Viết hàm tìm các chữ số xuất hiện nhiều nhất trong ma trận

22. Viết hàm liệt kê các cột có tổng nhỏ nhất trong ma trận

23. Viết hàm liệt kê các dòng có nhiều số hòan thiện nhất trong ma trận

24. Viết hàm hoán vị hai cột ma trận

25. Viết hàm hoán vị hai dòng ma trận

26. Viết hàm tính tổng, tích 2 ma trận

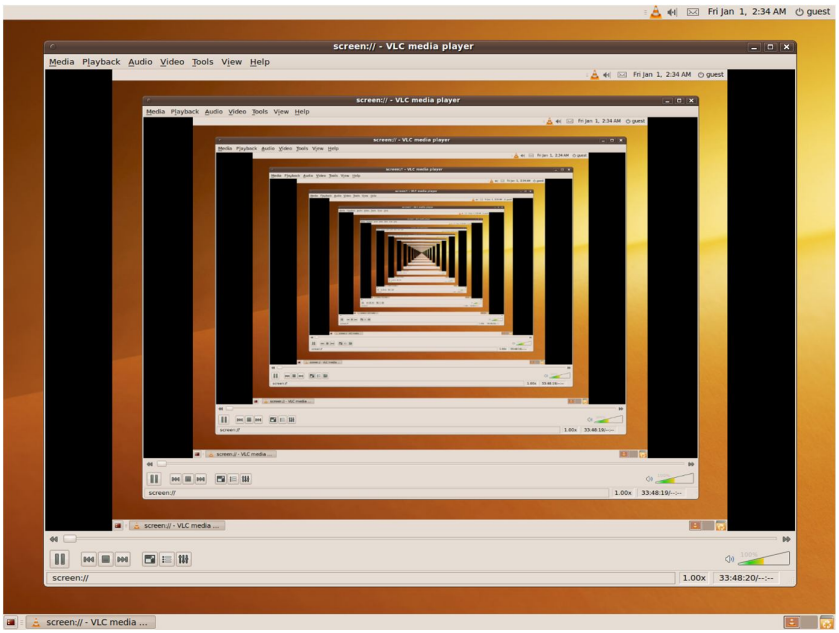
27. Viết hàm tìm định thức ma trận
28. Viết hàm sắp xếp theo yêu cầu: Dòng có chỉ số chẵn tăng dần, lẻ giảm dần
29. Viết hàm sắp xếp âm tăng dần, dương giảm dần, 0 giữ nguyên
30. Viết hàm sắp xếp các phần tử của ma trận theo hình ziczac

CHƯƠNG 3. KỸ THUẬT ĐỆ QUY

3.1 Khái niệm

Đệ quy (recursion) là một kỹ thuật được sử dụng trong các ngôn ngữ lập trình cao cấp như C/C++, C#, Java và ngày nay hầu hết các ngôn ngữ lập trình đều hỗ trợ kỹ thuật này. Đệ quy là kỹ thuật đưa bài toán hiện tại về một bài toán cùng loại, cùng tính chất nhưng ở cấp độ thấp hơn. Quá trình này tiếp tục cho đến khi bài toán được đưa về một cấp độ mà tại đó có thể giải được. Từ cấp độ này ta lần ngược trở lại để giải các bài toán ở cấp độ cao hơn cho đến khi giải xong bài toán ban đầu.

Hình 3.1 minh họa bản chất của kỹ thuật đệ quy, một ảnh chứa ảnh của chính nó nhưng ở kích thước nhỏ hơn, ảnh chứa ảnh đến khi ảnh còn là 1 điểm ảnh thì quá trình đệ quy dừng.



Hình 3.1: Ảnh minh họa kỹ thuật đệ quy

Trong toán học chúng ta cũng gặp các định nghĩa đệ quy, chẳng hạn tính giai thừa của một nguyên n.

Giai thừa của n (n!): Nếu n=0 thì n!=1; nếu n>0 thì n! = (n-1)! *n

Vậy một hàm được gọi là có tính đệ quy nếu trong bản thân hàm đó có lệnh gọi lại chính nó một cách trực tiếp hay gián tiếp nhưng ở cấp độ thấp hơn. Một chương trình được gọi là có tính đệ quy nếu nó có chứa ít nhất một hàm đệ quy.

Một hàm đệ quy gồm có 2 phần: phần cơ sở (hay còn gọi là phần neo, phần suy biến hoặc phần dừng) và phần đệ quy. Phần cơ sở thực hiện khi mà công việc giải bài toán quá đơn giản, có thể giải trực tiếp, phần này rất quan trọng vì nó quyết định tính hữu hạn dừng của hàm đệ quy; phần đệ quy là phần gọi lại chính nó nhưng với cấp độ thấp hơn.

Trong lập trình đệ quy, bản chất của nó là khi một hàm gọi đệ quy đến chính hàm đó, thì ở mỗi lần gọi tới, chương trình sẽ tạo ra một tập các biến cục bộ hoàn toàn độc lập với các tập biến cục bộ đã được tạo ra trong các lần gọi trước đó, có bao nhiêu lần gọi tới hàm đệ quy thì cũng có bấy nhiêu lần thoát khỏi hàm. Mỗi lần

thoát ra khỏi hàm thì một tập các biến cục bộ tương ứng của lần gọi đệ quy đó sẽ được giải phóng. Sự tương ứng giữa các lần gọi tới hàm và thoát ra khỏi hàm được thực hiện theo thứ tự ngược, nghĩa là lần thoát ra hàm đầu tiên tương ứng với lần gọi vào hàm sau cùng, và lần ra khỏi hàm sau cùng tương ứng lần đầu tiên gọi tới hàm.

Ví dụ 3.1:

Xét hàm đệ quy tính giai thừa của n

```
long tinhGiaiThua (int n)
{
  if(n==0)            //đây phần cơ sở của hàm đệ quy
       return 1;      //việc giải bài toán rất đơn giản.

  return tinhGiaiThua(n-1)*n; //đây là phần đệ quy của hàm
đệ quy
}       //nó gọi lại hàm tinhGiaiThua nhưng dành cho tham số n-1.
```

Kỹ thuật đệ quy có nhiều dạng vận dụng trong chương trình. Chúng ta sẽ tìm hiểu một số dạng phổ biến của kỹ thuật đệ quy: đệ quy tuyến tính, đệ quy đuôi, đệ quy nhị phân, đệ quy mũ, đệ quy lồng và đệ quy tương hỗ.

3.2 Các dạng đệ quy

3.2.1 Đệ quy tuyến tính (Linear Recursion)

Một hàm gọi là đệ quy tuyến tính (hay đệ quy đơn) nếu một lần gọi hàm nó chỉ phát sinh tối đa một lời gọi đệ quy.

Cấu trúc hàm dạng đệ quy tuyến tính:

<kiểu dữ liệu> <tên hàm> (<danh sách tham số>)

{

if (<điều kiện dừng cho phần cơ sở>)

{

[khối lệnh] //nếu có

return <giá trị trả về>;

}

[khối lệnh] //nếu có

<tên hàm>(<danh sách tham số>); //lệnh gọi hàm đệ qui

[khối lệnh] //nếu có

return <giá trị trả về>;

}

Ví dụ 3.2:

Hàm đệ quy tính cho biểu thức S(n) = 1*2 + 2*3 + … + (n-1)*n, với n>=1.

```
long tinhSn (int n)
 {
     if(n==1)            //phần cơ sở
```

```
            return 1*2;
        //phần đệ quy, chỉ gọi hàm đệ quy 1 lần
        return tinhSn(n-1) + (n-1)*n;
    }
```

Trong hàm đệ quy tính giai thừa của n ở ví dụ 3.2, ta thấy rõ ràng chỉ có 1 lời gọi đệ quy dành cho tham số (n-1) khi hàm được gọi. Tuy nhiên, trong hàm đệ quy tuyến tính, lời gọi hàm đệ quy cũng có thể xuất hiện nhiều hơn một lần trong thân hàm, nhưng khi thực thi hàm cũng chỉ gọi đệ quy một lần, chẳng hạn như hàm đệ quy ở ví dụ 3.3 sau đây.

Ví dụ 3.3:

Hàm tìm kiếm số x trên mảng 1 chiều a chứa các phần tử là số nguyên (mảng a đã có thứ tự tăng dần) bằng giải thuật tìm kiếm nhị phân. Hàm viết theo kỹ thuật đệ quy như sau:

```
int binarySearch (int a[], int left, int right, int x)
 {
     if(left<=right)
     {
          int mid=(left+right)/2;
     if(x==a[i])//phần cơ sở, điểm dừng của hàm đệ quy.
          return i;      //trả về vị trí x trong mảng a.

     if(x<a[i])
     return searchBinary(a, left, mid-1, x);//phần đệ quy

return searchBinary(a, mid+1, right ,x);//phần đệ quy
     }
     return -1;
}
```

Trong hàm tìm kiếm nhị phân trên, ta thấy có 2 phần cơ sở và 2 lần gọi đệ quy với giá trị tham số khác nhau. Tuy nhiên mỗi lần hàm thực thi, nó không thể thực hiện hết 2 lệnh dành cho phần cơ sở và 2 lệnh dành cho phần đệ quy. Vì khi hàm thực thi, tùy vào giá trị left so với right và x thuộc bên trái hay bên phải của phần tử a[mid] mà hàm sẽ gọi 1 trong 2 phần cơ sở, hoặc 1 trong 2 phần đệ quy. Do đó xét cho cùng, hàm tìm kiếm trên vẫn thuộc loại đệ quy tuyến tính.

3.2.2 Đệ quy nhị phân (Binary Recursion)

Một hàm được gọi là đệ quy nhị phân nếu mỗi lần gọi hàm nó phát sinh tối đa hai lần gọi đệ quy đến chính nó.

Cấu trúc hàm dạng đệ quy nhị phân:

<kiểu dữ liệu><tên hàm> (<danh sách tham số>)

{

if(<điều kiện dừng cho phần cơ sở>)

{

[khối lệnh] //nếu có

return <giá trị trả về>;

}

[khối lệnh] //nếu có

<tên hàm>(<danh sách tham số>) ; //lời gọi hàm đệ qui lần 1

<tên hàm>(<danh sách tham số>) ; //lời gọi hàm đệ qui lần 2

[khối lệnh] //nếu có

return <giá trị trả về>;

}

Ví dụ 3.4:

Hàm đệ quy tính số các tổ hợp chập k của n phần tử bằng đệ quy, biết rằng:

$$C_n^k = \begin{cases} 1, nếu\ k = 0\ hoặc\ k = n \\ C_{n-1}^{k-1} + C_{n-1}^{k}\ nếu\ 0 < k < n \end{cases}$$

```
int toHop(int n, int k)
{
     if (k == 0 || k == n)
          return 1; //phần cơ sở

     return toHop(n-1, k-1) + toHop(n-1, k); //phần đệ
qui có 2 lần gọi
     //hàm đệ qui
}
```

Trong hàm đệ qui ở ví dụ 3.4, ta thấy phần đệ qui có 2 lần gọi hàm đệ quy với giá trị tham số khác nhau. Khi thực thi, hàm gọi đệ qui đến 2 lần với các giá trị tham số khác nhau. Do dó, kết quả đầu ra của một lần gọi hàm đệ qui trên dựa vào kết quả trả về của 2 hàm đệ qui được gọi trong nó.

3.2.3 Đệ quy phi tuyến (NonLinear Recursion)

Một hàm gọi là đệ quy phi tuyến nếu bên trong thân hàm có lời gọi đệ quy được đặt bên trong thân vòng lòng.

Cấu trúc hàm dạng đệ qui phi tuyến

<kiểu dữ liệu><tên hàm> (<danh sách tham số>)

{

if(<điều kiện dừng cho phần cơ sở>)

{

[khối lệnh] //nếu có

return <giá trị trả về>;

}

[khối lệnh] //nếu có

vòng lặp (<điều kiện lặp>)

{

[khối lệnh] //nếu có

<tên hàm>(<danh sách tham số>) ; //lời gọi hàm đệ qui

[khối lệnh] //nếu có

}

[khối lệnh] //nếu có

return <giá trị trả về>;

}

Ví dụ 3.5:

Viết hàm tính U(n) theo công thức sau:

$$U(n) = \begin{cases} n, nếu\ n < 6 \\ U(n-5) + U(n-4) + U(n-3) + U(n-2) + U(n-1), nếu\ n \geq 6 \end{cases}$$

```
int TinhUn(int n)
{
     if(n<6)
          return n;
     int S=0;

     for(int i=1; i<6; i++)
          S=S+TinhUn(n-i);
     return S;
}
```

Ví dụ 3.6:

Viết hàm xuất các hoán vị phần tử mảng 1 chiều a chứa n số nguyên.

```
void xuatHoanViM1C (int a [], int n, int i)
{
     // n là số phần tử của a, i là số phần tử giữ lại
không hoán vị
     //Xuất mảng a
     for(int k=0; k<n; k++)
          cout<<a[i]<< “          ”;
     cout<<endl;
     //Thực hiện hoán vị
     for (j=i+1; j<n; j++)
     {
          int  swap = a[i];
          a[i] = a[j];
          a[j] = swap;
          xuatHoanViM1C (a, n, i+1);
     }
}
```

Trong hàm đệ qui phi tuyến, vì hàm gọi đệ quy được đặt trong vòng lặp nên số lần gọi đệ qui có thể chúng ta khó xác định cho mỗi lần gọi hàm. Chẳng hàn trong ví dụ 3.5, hàm gọi 5 lần đệ qui, nhưng trong ví dụ 3.6, số lần gọi đệ qui của hàm phụ thuộc giá trị n, i với i có sự thay đổi sau mỗi lần được gọi lại.

3.2.4 Đệ quy lồng (Nested Recursion)

Hàm được gọi là đệ qui lồng nếu tham số trong lời gọi hàm là một lời gọi đệ quy.

Cấu trúc hàm dạng đệ qui lồng

<kiểu dữ liệu><tên hàm> (<danh sách tham số>)

{

if(<điều kiện dừng cho phần cơ sở>)

{

[khối lệnh] //nếu có

return <giá trị trả về>;

}

[khối lệnh] //nếu có

//lời gọi hàm đệ qui có tham số là hàm gọi đệ qui

<tên hàm>(<tên hàm>(<danh sách tham số>), tham số) ;

[khối lệnh] //nếu có

return <giá trị trả về>;

}

Ví dụ 3.7:

Viết hàm tính giá trị Ackermann's theo công thức sau:

$$Acker(m,n) = \begin{cases} n+1, nếu\ m = 0 \\ Acker(m-1,1), nếu\ n = 0\ và\ m > 0 \\ Acker(m-1, Acker(m, n-1), nếu\ m > 0\ và\ n > 0 \end{cases}$$

Hàm số Ackermann là một hàm thực được mang tên nhà toán học người Đức Wilhelm Ackermann (1896–1962). Hàm Ackermann đôi khi còn được gọi là hàm Ackermann-Peter.

```
int ackerman (int m, int n)
{
     if (m == 0)
          return (n+1);
     if (n == 0)
          return ackerman(m-1,1);
     return ackerman(m-1, ackerman(m,n-1));
}
```

Trong ví dụ 3.7 có chứa lời gọi hàm đệ qui cho trường hợp m≠0 và n≠0, với tham số là kết quả trả về của 1 lời gọi hàm đệ qui của chính nó. Vì tính chất gọi lồng nhau nên sự phát triển của hàm đệ qui rất nhanh. Chẳng hạn nếu chúng ta phân tích quá trình xuất kết quả của hàm ackerman (gọi tắt là hàm A(m,n) thì giá trị cụ thể như sau:

m\n	0	1	2	3	4	n
0	0+1	1+1	2+1	3+1	4+1	$n+1$
1	A(0,1)	A(0,A(1,0))	A(0,A(1,1))	A(0,A(1,2))	A(0,A(1,3))	$n+2=2+(n+3)-3$
2	A(1,1)	A(1,A(2,0))	A(1,A(2,1))	A(1,A(2,2))	A(1,A(2,3))	$2n+3=2\cdot(n+3)-3$
3	A(2,1)	A(2,A(3,0))	A(2,A(3,1))	A(2,A(3,2))	A(2,A(3,3))	$2^{(n+3)}-3$
4	A(3,1)	A(3,A(4,0))	A(3,A(4,1))	A(3,A(4,2))	A(3,A(4,3))	$\underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{n+3}-3$
5	A(4,1)	A(4,A(5,0))	A(4,A(5,1))	A(4,A(5,2))	A(4,A(5,3))	*A*(4, *A*(5, n-1))
6	A(5,1)	A(5,A(6,0))	A(5,A(6,1))	A(5,A(6,2))	A(5,A(6,3))	*A*(5, *A*(6, n-1))

Nếu ta tính bằng tay cho hàm A(1,2) và A(4,3) thì kết quả sẽ có sự biến đổi rất lớn, trãi qua rất nhiều bước, cụ thể như sau:

$$\begin{aligned} A(1,2) &= A(0,A(1,1)) \\ &= A(0,A(0,A(1,0))) \\ &= A(0,A(0,A(0,1))) \\ &= A(0,A(0,2)) \\ &= A(0,3) \\ &= 4. \end{aligned}$$

$$
\begin{aligned}
A(4,3) &= A(3,A(4,2)) \\
&= A(3,A(3,A(4,1))) \\
&= A(3,A(3,A(3,A(4,0)))) \\
&= A(3,A(3,A(3,A(3,1)))) \\
&= A(3,A(3,A(3,A(2,A(3,0))))) \\
&= A(3,A(3,A(3,A(2,A(2,1))))) \\
&= A(3,A(3,A(3,A(2,A(1,A(2,0)))))) \\
&= A(3,A(3,A(3,A(2,A(1,A(1,1)))))) \\
&= A(3,A(3,A(3,A(2,A(1,A(0,A(1,0))))))) \\
&= A(3,A(3,A(3,A(2,A(1,A(0,A(0,1))))))) \\
&= A(3,A(3,A(3,A(2,A(1,A(0,2)))))) \\
&= A(3,A(3,A(3,A(2,A(1,3))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(1,2)))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(0,A(1,1))))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(0,A(0,A(1,0)))))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(0,A(0,A(0,1)))))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(0,A(0,2)))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(0,3)))))) \\
&= A(3,A(3,A(3,A(2,A(0,4))))) \\
&= A(3,A(3,A(3,A(2,5)))) \\
&= \dots \\
&= A(3,A(3,A(3,13))) \\
&= \dots \\
&= A(3,A(3,65533)) \\
&= \dots \\
&= A(3,2^{65536}-3) \\
&= \dots \\
&= 2^{2^{65536}}-3.
\end{aligned}
$$

3.2.5 Đệ quy tương hỗ (Mutual Recursion)

Trong hàm đệ qui tương hỗ, phần đệ quy không cần thiết phải gọi chính nó mà có thể gọi một hàm đệ qui khác. Đệ quy tương hỗ thì thường có 2 hàm, và trong thân của hàm này có lời gọi của hàm kia , điều kiện dừng và giá trị trả về của cả hai hàm có thể giống nhau hoặc khác nhau.

Cấu trúc đệ qui tương hỗ:

```
<kiểu dữ liệu> <tên hàm X> (<danh sách tham số>)
{
   if(<điều kiện dừng cho phần cơ sở>)
   {
         [khối lệnh] //nếu có
```

```
        return <Gia tri tra ve>;
    }
    [khối lệnh] //nếu có
    return <tên hàm X>(<danh sách tham số>)<liên kết 2 hàm> <tên hàm
    Y>(<danh sách tham số>) ;
}

<kiểu dữ liệu> <tên hàm Y> (<danh sách tham số>)
{
    if(<điều kiện dừng cho phần cơ sở>)
    {
        [khối lệnh] //nếu có
        return <Gia tri tra ve>;
    }
    [khối lệnh] //nếu có
    return <tên hàm Y>(<danh sách tham số>)<liên kết 2 hàm> <tên hàm
    X>(<danh sách tham số>) ;
}
```

Ví dụ 3.8 :

Viết hàm tính 2 biểu thức F(n) và G(n) theo công thức sau :

$$F(n) = \begin{cases} n, nếu\ n < 5 \\ F(n-1) + G(n-2), nếu\ n \geq 5 \end{cases}$$

$$G(n) = \begin{cases} n-3, nếu\ n < 8 \\ F(n-1) + G(n-2), nếu\ n \geq 8 \end{cases}$$

```
long F ( int n)
{
      if (n<5)
          return n;
      return F(n-1) + G(n-2);
}

long G (int n)
{
     if (n<8)
          return n-3;
      return F(n-1) + G(n-2);
}
```

Trong ví dụ 3.8, hàm F và G gọi đệ qui lẫn nhau, vì vậy khi tính F cần có tính G và ngược lại.

3.2.6 Những ưu nhược điểm của kỹ thuật đệ quy

Kỹ thuật đệ quy là một trong những kỹ thuật lập trình thông dụng và thường được dùng phổ biến ở nhiều chương trình lập trình. Tuy nhiên tùy thuộc vào mỗi loại bài toán mà kỹ thuật này có những ưu điểm và nhược điểm riêng, cho nên có những bài toán cần phải dùng đệ quy để giải quyết đơn giản và ngược lại có những bài toán không nên dùng đệ quy vì sẽ làm cho chương trình chạy châm và tốn nhiều bộ nhớ.

- Ưu điểm :

+ Viết chương trình đơn giản dễ hiểu.

+ Có thể thực hiện một số lượng lớn các thao tác tính toán thông qua một đoạn chương trình ngắn gọn.

+ Định nghĩa một tập vô hạn các đối tượng thông qua một số hữu hạn lời phát biểu.

+ Hầu hết các ngôn ngữ lập trình đều hỗ trợ kỹ thuật đệ quy.

+ Đưa ra phương án tối ưu, giải quyết được một số vấn đề mà vòng lặp thông thường không giải quyết được.

- Nhược điểm

+ Do mỗi lần gọi hàm đệ qui, chương trình phát sinh vùng nhớ mới trong bộ nhớ nên dễ tốn bộ nhớ.

+ Chương trình có tốc độ chạy chậm.

3.3 Các bước tìm giải thuật đệ quy cho một bài toán

Để xây dựng giải thuật giải một bài toán có tính đệ quy bằng phương pháp đệ quy, ta cần thực hiện tuần tự 3 bước chính sau :

- Thông số hóa bài toán.
- Tìm các trường hợp cơ bản (phần cơ sở) cùng giải thuật tương ứng cho các trường hợp này.
- Tìm giải thuật giải trong trường hợp tổng quát (phần đệ quy) bằng cách phân rã bài toán theo kiểu đệ quy.

3.3.1 Thông số hóa bài toán

Tổng quát hóa bài toán cụ thể cần giải thành bài toán tổng quát (một họ các bài toán chứa bài toán cần giải), tìm ra các thông số cho bài toán tổng quát đặc biệt là nhóm các thông số biểu thị kích thước của bài toán – các thông số điều khiển (các thông số mà độ lớn của chúng đặc trưng cho độ phức tạp của bài toán, và giảm đi sau mỗi lần gọi đệ qui).

Ví dụ 3.9: Tham số n trong hàm tinhGiaiThua ở ví dụ 3.1 ; tham số m,n trong hàm Ackerman ở ví dụ 3.7.

3.3.2 Tìm các trường hợp cơ bản (phần cơ sở) cùng giải thuật tương ứng cho các trường hợp này.

Đây là trường hợp suy biến của bài toán tổng quát, là các trường hợp tương ứng với các giá trị biên của các biến điều khiển (dành cho trường hợp bài toán có giá trị tham số nhỏ nhất) mà giải thuật giải không đệ quy. Thông thường trong trường hợp này giải thuật rất đơn giản.

Ví dụ 3.10 :

- Giá trị giai thừa của n=0 hoặc n=1 là 1 (ví dụ 3.1)
- Giá trị hàm U(n)= n, với n<6 (ví dụ 3.5)
- Giá trị hàm Ackerman = n+1, với m=0 và n>0 (ví dụ 3.8)

3.3.3 Phân rã bài toán tổng quát theo phương thức đệ quy

Tìm giải thuật giải bài toán trong trường hợp tổng quát bằng cách phân chia nó thàn các thành phần hoặc có giải thuật không đệ quy hoặc là bài toán trên nhưng có kích thước nhỏ hơn.

Mục đích của bước thực hiện này là tìm giải thuật để giải bài toán theo hướng đệ qui với giá trị tham số tổng quát.

Ví dụ 3.11

- Tính giai thừa số nguyên n: n !=(n-1) ! * n, với n>=2
- Tính tổ hợp chập k của n : $C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$ nếu $0 < k < n$

3.4 Một số bài toán đệ quy thông dụng

3.4.1 Bài toán tìm tất cả hoán vị của một dãy phần tử.

Yêu cầu bài toán : xuất tất cả các hoán vị của dãy A.

- Nếu dãy A có n=1 phần tử A[1] = a thì số hoán vị chỉ có 1 là : a.
- Nếu dãy A có n=2 phần tử : A[1]=a, A[2]=b thì số hoán vị là 2 dãy sau : a b ; b a.
- Nếu dãy A có n=3 phần tử : A[1]=a, A[2]=b, A[3]=c thì các hoán vị của dãy A là 6 dãy sau:

a b c ; a c b

b a c ; b c a

c b a ; c a b

- Vậy với dãy A gồm n=4 phần tử : A[1]=a, A[2]=b, A[3]=c, A[4]=d, thì các dãy hoán vị của dãy A là 24 dãy sau :

a b c d	a b d c	a d b c	a d c b
a c d b	a c b d	b a c d	b a d c
b c d a	b c a d	b d c a	b d a c
c b a d	c b d a	c d b a	c d a b
c a d b	c a b d	d a c b	d a b c
d c b a	d c a b	d b c a	d b a c

3.4.1.1Thông số hóa bài toán

Gọi hàm HoanVi(A, m) (với A : array[1..N] của T, m : integer, T là một kiểu dữ liệu đã biết trước) là thủ tục xuất tất cả các dạng khác nhau của A có được bằng cách hoán vị m thành phần đầu của dãy A.

Ví dụ 3.12 :

Với N=4, mảng A có : A[1]=a, A[2]=b, A[3]=c, A[4]=d thì lời gọi HoanVi (A,3) xuất tất cả hoán vị của A có được bằng cách hoán vị 3 phần tử đầu (có 6 dãy hoán vị) :

a b c d b a c d c a b d

a c b d b c a d c b a d

Vậy để giải bài toán đặt ra ban đầu ta gọi hàm HoanVi(A, N).

3.4.1.2Tìm phần cơ sở và cách giải.

Xét lại hàm HoanVi(A,m), ta thấy rằng:

Nếu m=1 : hàm HoanVi(A,1) xuất tất cả các dạng của v có được bằng cách hoán vị 1 phần tử đầu. Vậy HoanVi(A,1) chỉ có 1 cách là xuất dãy A.

Code xử lý cho HoanVi(A,1) là xuất dãy A :

```
for(int i=0 ; i< N ; i++)       // trong lập trình, các phần tử mảng tính từ vị trí 0
    cout<<A[i]
```

3.4.1.3Phân rã bài toán trong trường hợp tổng quát.

Phân tích các phần tử của dãy A : A[1], A[2], … , A[m-1], A[m], A[m+1],…, A[N].

Ta có thể tìm hết tất cả các hoán vị của m phần tử đầu của dãy A theo cách phân tích sau :

- Hoán vị (m-1) phần tử đầu (gọi hàm đệ quy HoanVi(A,m-1), giữ nguyên các phần tử cuối A[m],…, A[N].
- Đổi chỗ A[m] cho A[m-1] ; (m-1) phần tử đầu (gọi hàm đệ quy HoanVi(A,m-1), giữ nguyên các phần tử cuối A[m],…, A[N].
- Đổi chỗ A[m] cho A[m-2] ; (m-1) phần tử đầu (gọi hàm đệ quy HoanVi(A,m-1), giữ nguyên các phần tử cuối A[m],…, A[N].
- …
- Đổi chỗ A[m] cho A[2] ; (m-1) phần tử đầu (gọi hàm đệ quy HoanVi(A,m-1), giữ nguyên các phần tử cuối A[m],…, A[N].
- Đổi chỗ A[m] cho A[1] ; (m-1) phần tử đầu (gọi hàm đệ quy HoanVi(A,m-1), giữ nguyên các phần tử cuối A[m],…, A[N].

Gọi hàm Swap (x,y) là thủ tục để hoán đổi giá trị 2 đối tượng x và y cho nhau.

Vậy $HoanVi(A, m) \equiv$ { Swap(A[m], A[m]) ; HoanVi(A,m-1) ;

Swap(A[m], A[m-1] ; HoanVi(A,m-1) ;

Swap(A[m], A[m-2] ; HoanVi(A,m-1) ;

…

Swap(A[m], A[2] ; HoanVi(A,m-1) ;

Swap(A[m], A[1] ; HoanVi(A,m-1) ;

}

tương đương

HoanVi(A,m) $\equiv for(int\ i = m\ ; i \geq 1\ ; i--$

{

Swap(A[m], A[i] ; HoanVi(A,m-1) ;

```
}
```

3.4.1.4Chương trình mã hóa giải thuật Hoán vị bằng ngôn ngữ lập trình C/C++ :

```
const N = val;// val là hằng giá trị
Data A[N]; //mang A chứa dãy dữ liệu với kiểu dữ liệu Data
//.............
void Swap( Data &x, Data &y)
{
        Data t=x;
        x=y;
        y=t;
}
//.............
void print (Data A[])// mảng A có N phần tử
{
        for(int i=0; i<N; i++)          // phần tử mảng bắt đầu từ vị trí 0
                cout<<A[i];
}
//.............
void HoanVi(Data A[], int m)// hoán vị m thành phần đầu của dãy A
{
        if(m==1)
                print(A)
        else
                for(int i=m-1; i>=0; i--)// phần tử mảng bắt đầu từ vị trí 0
                {
                        Swap(A[m-1],A[i]);
                        HoanVi(A,m-1)
                }
}
```

3.4.2 Bài toán sắp xếp mảng bằng phương pháp trộn (Merge Sort)

Mô tả bài toán : Để sắp xếp tăng/giảm 1 danh sách gồm n phần tử bằng phương pháp trộn người ta chia danh sách thành 2 phần (tổng quát là nhiều phần), sắp xếp từng phần rồi trộn chúng lại với nhau theo thứ tự.

3.4.2.1. Thông số hóa

Bài toán được khái quát thành sắp xếp một dãy con (từ vị trí thứ left đến vị trí thứ right, $1 \le left \le right \le n$) của dãy A có n phần tử. Gọi thủ tục sắp xếp là MergeSort (A, left, right).

Nếu sắp xếp cả dãy A thì ta có lời gọi hàm MergeSort(A,1,n).

3.4.2.2. Tìm phần cơ sở và cách giải.

Nếu dãy A chỉ có 1 phần tử (n=1), khi đó không cần làm gì cả, thao tác rỗng.

3.4.2.3. Phân rã bài toán trong trường hợp tổng quát.

Khi left<right, cần thực hiện các công việc sau :

- Chia dãy : A[left], A[left+1],…, A[right] thành 2 dãy con

A[left], A[left+1],…, A[mid] và A[mid+1], A[mid+2], …., A[right].

- Sắp xếp từng dãy con thành các dãy có thứ tự theo giải thuật Merge Sort.
- Trộn 2 dãy con có thứ tự lại thành dãy A[left]…A[right] mới có thứ tự.

Để thực hiện trộn 2 dãy có thứ tự thành một dãy có thứ tự ta sẽ dùng thủ tục không đệ quy Merge (A, left, mid, right). Ta cần chọn s để được 2 dãy con giảm hẳn kích thước so với dãy ban đầu, tức là chọn s : $left < mid < mid + 1 < right$. Thông thường chọn mid là vị trí ở giữa dãy : mid = (left+right)/2.

3.4.2.4. Chương trình mã hóa giải thuật MergeSort bằng ngôn ngữ lập trình C:

```
const N = val;// val là hằng giá trị
Data A[N]; //mang A chứa dãy dữ liệu với kiểu dữ liệu Data
//……..
void Merge(Data A[], int left, int mid, int right)
{
     Data temp_arr[N]; //mảng lưu tạm dữ liệu của A
     int index = left, saveFirst = left;

     //Đặt lại các biến đầu cuối của 2 dãy con
     int left1 = left;
     int right1 = mid;
     int left2 = mid+1;
     int right2 = right;

     while((left1 <= right1)&&(left2 <=right2))
     {
          if(A[left1] < A[left2]) //đưa giá trị nhỏ nhất của 2
dãy vào mảng tạm
          {
               temp_arr[index] = A[left1];
               left1 ++;
          }
          else
          {
               temp_arr[index] = A[left2];
               left2 ++;
          }
          index ++;
     }
```

```
     // Nếu 1 trong 2 dãy còn lại chưa đưa vô mảng tạm thì lần
lượt đưa các
     //phần tử còn lại vào mảng tạm có thứ tự

     while(left1 <= right1)
     {
          temp_arr[index] = A[left1];
          left1 ++;
          index ++;
     }
     while(left2 <= right2)
     {
          temp_arr[index] = A[left2];
          left2 ++;
          index ++;
     }

     //Chép mảng tạm đã trộn 2 dãy con trả lại mảng A
     for(index = saveFirst; index <= right; index++)
          A[index] = temp_arr[index];
}

void MergeSort(Data A[], int left, int right)
{
     if(left<right)
     {
          int mid = (left+right)/2;
          MergeSort(A, left, mid);
          MergeSort(A, mid+1, right);
          Merge(A, left, mid, right);
     }
}
```

3.4.3 Bài toán chia thưởng

Có 100 phần thưởng đêm chia cho 12 học sinh giỏi đã được xếp hạng. Có bao nhiêu cách khác nhau để thực hiện cách chia ?

Việc tìm ra lời giải cho bài toán sẽ không dễ dàng nếu ta không tìm ra cách thích hợp để tiếp cận nó. Ta sẽ tìm giải thuật giải bài toán này theo phương pháp đệ quy.

3.4.3.1. Thông số hóa bài toán.

Phát biểu bài toán ở mức tổng quát : tìm số cách chia m vật (phần thưởng) cho n đối tượng (học sinh) có thứ tự.

- Gọi Distribute là hàm tính số cách chia, khi đó Distribute là hàm có 2 tham số nguyên m và n (Distribute(m, n)).
- Gọi n đối tượng theo thứ tự xếp hạng 1, 2, 3, …, n ; S_i là số vật mà đối tượng thứ i nhận được.

Khi đó các điều kiện ràng buộc cho cách chia là :

$S_i \geq 0$

$S_1 \geq S_2 \geq \cdots \geq S_n$

$S_1 + S_2 + \cdots + S_n = m$

Ví dụ :

Với m=5, n=3 có 5 cách chia như sau :

5 0 0

4 1 0

3 2 0

3 1 1

2 2 1

Vậy Distribute(5,3) = 5

3.4.3.2. Tìm phần cơ sở và cách giải

- Nếu m=0 (số vật bằng 0) thì sẽ có 1 cách chia : mọi đối tượng đều nhận 0 vật. Vậy Distribute(0, n) = 1 với mọi giá trị n.
- Nếu n=0, m≠ 0 thì không có cách nào để thực hiện việc phân chia. Vậy Distribute(m,0)=0 với mọi giá trị m≠ 0.
- Hoặc nếu n=1, m≠ 0 thì sẽ có 1 cách chia : đối tượng nhận m vật. Khi đó Distribute(m,1)=1.

3.4.3.3. Phân rã bài toán trong trường hợp tổng quát.

- Nếu m<n (số vật nhỏ hơn số đối tượng) thì n-m đối tượng xếp sau cuối sẽ không nhận được gì trong mọi cách chia. Vậy khi m<n thì Distribute(m,n)=Distribute(m,m).
- Nếu m ≥ n (số vật lớn hơn hoặc bằng số đối tượng) ta phân các cách chia thành 2 nhóm :

+ Nhóm 1 : không dành cho đối tượng thứ n (đối tượng xếp cuối) số vật nào cả ($S_n = 0$). Số cách chia này sẽ bằng số cách chia m vật cho (n-1) đối tượng. Vậy số cách chia trong nhóm 1 = Distribute (m,n-1).

+ Nhóm 2 : có phân chia vật cho người cuối cùng (Sn>0). Ta thấy rằng số cách chia của nhóm này bằng số cách chia (m-n) vật cho n đối tượng. Vì cách chia mà tất cả đối tượng đều nhận được phần thưởng có thể thực hiện bằng cách cho mỗi đối tượng nhận trước 1 vật, sau đó chia phần vật còn lại.

Do vậy số cách chia trong nhóm 2 = Distribute (m-n,n). Hay với $m \geq n$:

Distribute(m,n) = Distribute(m,n-1) + Distribute (m-n, n)

3.4.3.4. Chương trình mã hóa giải thuật chia thưởng bằng ngôn ngữ lập trình C:

```
int Distribute (int m, int n)
{
     if(m==0||n==1)
          return 1;
     if(n==0)
          return 0;
     if (m<n)
          return Distribute(m,m);
```

```
        return Distribute(m,n-1) + Distribute(m-n,n);
    }
```

3.4.4 Bài toán tháp Hà Nội

Truyền thuyết kể rằng : một nhà toán học Pháp sang thăm Đông Dương đến một ngôi chùa cổ ở Hà Nội thấy các vị sư đang chuyển một chồng đĩa quý gồm 64 đĩa với kích thước khác nhau từ cột A sang cột C theo cách :

- Mỗi lần di chuyển 1 đĩa.
- Khi chuyển có thể dùng cột trung gian B.
- Trong quá trình chuyển các chồng đĩa ở các cột luôn được xếp đúng (đĩa có kích thước bé đặt trên đĩa có kích thước lớn).

Khi được hỏi, các vị sư cho biết khi chuyển xong chồng đĩa thì sẽ đến ngày tận thế !.

Sau đó người ta chứng minh rằng với chồng n đĩa cần $2^n - 1$ lần chuyển cơ bản (chuyển một lần 1 đĩa).

Giả sử thời gian để chuyển 1 đĩa là t giây thì thời gian chuyển xong chồng 64 đĩa sẽ là : $T = (2^{64} - 1) * t\ (giây) = 1.84 * 10^{19} * t\ (giây)$

với t=1/100 (giây) thì $T=5.8*10^9$ năm = 5.8 tỷ năm.

Ta có thể tìm thấy giải thuật để giải bài toán trên một cách dễ dàng ứng với trường hợp chồng đĩa gồm : 0, 1, 2, 3 đĩa như sau :

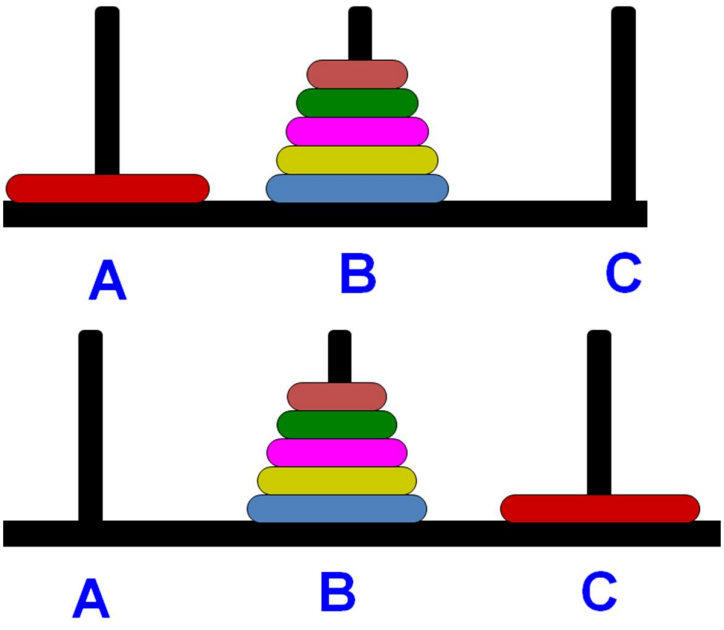
Chồng có 0 đĩa : 2^0-1= 0 lần chuyển

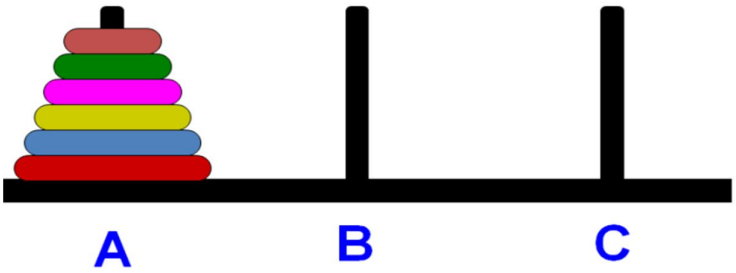
Chồng có 1 đĩa : 2^1-1=1 lần chuyển

Chồng có 2 đĩa : 2^2-1=3 lần chuyển

Chồng có 3 đĩa : 2^3-1=7 lần chuyển

Với chồng 4 đĩa, giải thuật bài toán trở nên phức tạp. Tuy nhiên giải thuật của bài toán lại được tìm thấy rất dễ dàng nhanh chóng khi ta khái quát số đĩa là n bất kỳ và nhìn bài toán bằng quan niệm đệ quy.





Hình mình họa quá trình giải tổng quát bài toán Tháp Hà Nội

3.4.4.1. Thông số hóa bài toán

Xét bài toán ở mức tổng quát : chuyển n đĩa từ cột A sang cột C có cột B làm trung gian.

Ta gọi giải thuật giải bài toán ở mức tổng quát là hàm HaNoiTower(n, A, B, C) chứa 4 tham số :

+ n (n>=0) : số nguyên dương, đại diện số đĩa cần di chuyển.

+ A, B, C : thuộc tập các ký tự, đại diện các cột chứa đĩa.

Bài toán mô tả ở trên sẽ được thực hiện bằng lời gọi HaNoiTower (64, A, B, C). Khi đó ta thấy rằng 4 tham số của bài toán thì tham số n quyết định độ phức tạp của bài toán. Vì n càng lớn thì số thao tác chuyển đĩa càng nhiều và thức tự thực hiện chúng càng khó hình dung. Do đó n là thông số điều khiển của bài toán.

3.4.4.2. Tìm phần cơ sở và cách giải

Khi n=1, bài toán tổng quát suy biến thành bài toán đơn giản HaNoiTower(1, A, B, C) : tìm dãy thao tác để chuyển chồng 1 đĩa từ cột A sang cột C, lấy cột B làm cột trung gian.

Gọi hàm di chuyển 1 đĩa giữa 2 cột bất kỳ là Move. Chẳng hạn di chuyển 1 đĩa từ cột A sang cột B là Move(A,B).

Giải thuật giải bài toán (1, A, B, C) : thực hiện 1 thao tác cơ bản đó là chuyển 1 đĩa từ cột A sang cột C (Move(A,C)).

Theo như qui ước trên ta có : $HaNoiTower(1, A, B, C) \equiv \{Move(A, C)\}$

Tương tự ta cũng có thể xác định trường hợp cho phần cơ sở khi n=0. Nó tương ứng bài toán HaNoiTower(0, A, B, C) : di chuyển 0 đĩa từ cột A sang cột C, lấy cột B làm trung gian. Khi đó giải thuật tương ứng thực hiện 0 thao tác, hoặc tập thao tác là rỗng : $HaNoiTower(A, B, C) \equiv \{\emptyset\}$.

3.4.4.3. Phân rã bài toán

Bài toán tháp Hà Nội có thể phân rã như sau: HaNoiTower(n, A, B, C): chuyển n đĩa từ cột A sang cột C, lấy cột B làm trung gian thành dãy tuần tự 3 công việc sau:

- Chuyển (n-1) đĩa từ cột A sang cột B, lấy cột C làm trung gian :

  HaNoiTower (n-1, A, C, B).

- Chuyển 1 đĩa từ cột A sang cột C :

  Move(A, C).

- Chuyển (n-1) đĩa từ cột B sang cột C, lấy A làm trung gian :

HaNoiTower (n-1, B, A, C)..

Vậy giải thuật tổng quát giải bài toán với n>1 hay phần đệ quy của giải thuật là :

$HaNoiTower(A, B, C) \equiv \{$ $Thap\_HN(n-1, A, C, B)$;

Move (A, C) ;

$Thap\_HN(n-1, B, A, C)$;$\}$

Do vây, với n đĩa thì cần bao nhiêu bước chuyển 1 đĩa ? Thực chất trong hàm HaNoiTower, các lệnh gọi đệ quy nhằm sắp xếp trình tự các thao tác chuyển 1 đĩa. Số lần chuyển 1 đĩa được thực hiện là đặc trưng cho độ phức tạp của giải thuật. Với n đĩa, gọi f(n) là số các thao tác chuyển 1 đĩa, ta có :

f(0) = 0.

f(1) = 1.

…

f(n) = 2f(n-1)+1, với n>1.

Do đó : f(n) = 1+2+ 2^2 + … + 2^{n-1} = 2^n – 1.

Để chuyển 64 đĩa cần $2^{64} - 1 \approx 2^{10}$ bước.

3.4.4.4. Chương trình mã hóa giải thuật HaNoiTower bằng ngôn ngữ lập trình C:

```
void HaNoiTower(int n, char A, char B, char C)
{
     if(n>0)
          {
               HaNoiTower(n-1, A, C, B);
               Move(A, C);
               HaNoiTower(n-1, B, A, C);
          }
     return; //nếu n==0 thì thực hiện 0 thao tác
}
```

hoặc :

```
void HaNoiTower(int n, char A, char B, char C)
{
     if(n==1)
          Move(A, C);
     else
     {
          HaNoiTower(n-1, A, C, B);
          Move(A, C);
          HaNoiTower(n-1, B, A, C);
     }
     return;
}
```

3.5 Khử đệ quy

Đặc điểm của quá trình xử lý một giải thuật đệ quy là khi thực thi lời gọi đệ quy, sinh ra lời gọi đệ quy mới cho đến khi gặp trường hợp suy biến (trường hợp neo). Cho nên để thực thi giải thuật đệ quy chương trình cần có cơ chế lưu trữ thông tin thỏa các yêu cầu sau :

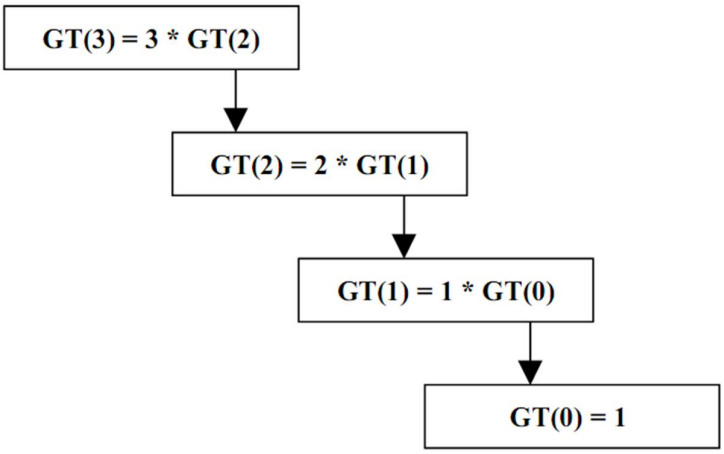
- Mỗi lần gọi hàm đệ quy mới phải lưu trữ thông tin trạng thái hàm con dang dở của tiến trình xử lý ở thời điểm gọi đệ quy. Số trạng thái được lưu trữ này bằng số lần gọi chưa được hoàn tất từ trước đó.
- Khi thực hiện xong mỗi lần gọi, cần khôi phục lại toàn bộ thông tin trạng thái trước khi gọi đệ quy.
- Lệnh gọi cuối cùng (ứng với trường hợp cơ sở) sẽ được hoàn tất đầu tiên. Thứ tự dãy các lệnh gọi được hoàn tất ngược với thứ tự gọi, tương ứng dãy thông tin trạng thái được phục hồi theo thứ tự ngược với thứ tự lưu trữ.

Ví dụ 3.13 :

Xét quá trình thực hiện đệ quy của hàm tính giai thừa số nguyên n.

```
long GT (int n)
{
  if(n==0)  return 1;
  return GT(n-1)*n;
}
```

Trong trường hợp n=3, ta có quá trình thực hiện của hàm như sau :



Khi thực hiện lời gọi GT(3) thì sẽ phát sinh lời gọi hàm đến GT(2) và đồng thời phải lưu giữ thông tin trạng thái xử lý còn dang dở GT(3) = 3 * GT(2). Đến lượt hàm GT(2) sẽ phát sinh lời gọi hàm đến GT(1) và lưu giữ thông tin trạng thái còn dang dở GT(2) = 2 * GT(1)… Quá trình cứ thực hiện tương tự cho tới khi gặp trường hợp suy biến GT(0) = 1.

Kết thúc quá trình gọi đệ quy là quá trình xử lý ngược được thực hiện:

- Giá trị của GT(0) được dùng để tính GT(1) theo quá trình lưu trữ.
- Dùng giá trị GT(1) để tính GT(2) theo quá trình tương tự.
- Dùng giá trị GT(2) để tính GT(3) để ra kết quả cuối cùng.
- Song song với quá trình xử lý ngược là xóa bỏ thông tin lưu trữ trong những lần gọi hàm tương ứng.

Đệ quy là phương pháp giúp chúng ta tìm giải thuật cho các bài toán khó. Giải thuật giải bài toán bằng đệ quy thường rất gọn gàng, dễ hiểu, dễ thể hiện chương trình bằng các ngôn ngữ lập trình. Nhưng việc dùng giải thuật đệ quy dễ dẫn đến phát sinh lưu trữ nhiều dữ liệu, tốn rất nhiều thời gian xử lý. Vì vậy việc tìm cách thay thế một chương trình đệ quy hoặc hàm đệ quy bằng một chương trình không đệ quy, hàm không đệ quy là một vấn đề cần thiết.

Khử đệ quy là quá trình chuyển đổi một giải thuật đệ quy thành 1 giải thuật không có đệ quy. Thực tế, vì hàm đệ quy rất đa dạng và nhiều loại, cho nên việc tìm giải thuật không đệ quy để thay thế cho các giải thuật đệ quy thì không đơn giản, và chưa có giải pháp chung cho việc chuyển đổi này một cách tổng quát. Cách tiếp cận thường dùng để khử đệ quy là dùng vòng lặp, hoặc dùng stack (cấu trúc ngăn xếp) để lưu các giá trị xử lý của bài toán. Tuy nhiên vẫn có những bài toán chúng ta không thể giải quyết nếu không dùng phương pháp đệ quy.

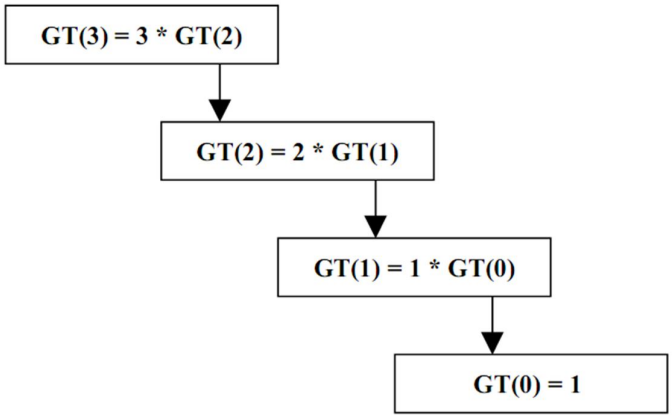
Trong nội dung giáo trình giới thiệu 2 cách khử đệ quy thông dụng dành cho các bài toán đệ quy có thể thực hiện bằng cách không đệ quy. Đó là khử đệ quy bằng vòng lặp và dùng cấu trúc dữ liệu Stack.

3.5.1 Khử đệ quy đơn giản bằng vòng lặp.

Ý tưởng của phương pháp là dùng vòng lặp để lưu lại giá trị của các lần tính toán trước làm dữ liệu cho việc tính toán của lần sau.

Ví dụ 3.13 :

Xét hàm đệ quy và khử đệ quy trong bài toán tính giai thừa.




```
//Hàm tính giai thừa của số nguyên n bằng kỹ thuật đệ quy

long GT (int n)

{

      if(n==0)        return 1;

      else            return  n * GT(n-1);

}

//Hàm tính giai thừa của số nguyên n bằng kỹ thuật không đệ quy

long GT( int n)

{  long K=1;

  for (int i =2; i<=n;i++) K=K*i;

  return K;

}
```

Điều kiện biên

Giá trị cần lưu

K là kết quả của giá trị giai thừa trước đó

3.5.2 Khử đệ quy dùng stack.

Để thực hiện một hàm đệ quy thì hệ thống phải tổ chức vùng nhớ lưu trữ theo quy tắc LIFO (Last In First Out – cơ chế vào sau ra trước của cấu trúc dữ liệu Ngăn xếp (Stack)). Hầu hết các ngôn ngữ lập trình cấp cao đều có khả năng tạo vùng nhớ stack mới cho phép tổ chức các chương trình đệ quy.

Nếu thực hiện một hàm đệ quy theo cách mặc định thường tốn bộ nhớ. Do cách tổ chức stack mặc định thích hợp cho mọi trường hợp nên sẽ không tối ưu trong từng trường hợp cụ thể. Do đó sẽ tốt khi người lập trình chủ động tạo cấu trúc dữ liệu stack đặc dụng cho từng hàm đệ quy cụ thể.

Giả sử thủ tục đệ quy tuyến tính F(X) có cấu trúc như sau :

```
F(X)
{
   if(dk(X))       //phần neo
          A(X);
   B(X);
   F(f(X));
   C(X);
}
```

Trong đó:

F(X): hàm đệ qui có danh sách tham số là X.

dk(X): biểu thức điều kiện theo X.

A(X), B(X), C(X): nhóm lệnh không đệ quy theo X.

f(X): hàm thay đổi X để gọi đệ quy cấp thấp hơn.

Giải thuật thực hiện F(X) với việc khử đệ quy bằng cách dùng Stack có dạng như sau:

```
F(X)
{
    taoStack (S);
        //đưa các giá trị đã tính vào Stack
        while(dk(X) == false)
        {
            B(X);
            Push(S,Xgt);  //Xgt là giá trị cần lưu trữ sau mỗi bước tính với X.
            X = f(X)
        }
        A(X);
        //Lần lượt lấy các giá trị trong Stack ra.
        while( Empty(S) == flase)
        {
            Pop(S,Xgt);
            C(X);
        }
}
```

Ví dụ 3.14:

Xét hàm đệ quy chuyển số n dạng thập phân sang số dạng nhị phân.

```
void binary (int n)
{
  if(n>0)
  {
        binary(n/2);
        cout<<n%2;
  }
}
```

Trong đó:

- X là n.
- F(X) là binary(X).
- A(X), B(X) là không có.
- C(X) là lệnh xuất n%2.
- dk(X) là n<=0.
- f(X) là f(n) = n/2.

Giải thuật khử đệ hàm binary bằng cách dùng Stack như sau:

```
void binary (int n)
{
   int temp;
   taoStack (S);
   while(n>0)
   {
         temp = n%2;
         Push(S,temp);
         n/=2;
   }
   while(Empty(S)==false)
   {
         Pop(S,temp);
         cout<<temp;
   }
}
```

Giải thuật đệ quy nhị phân F(X) có dạng:

```
F(X)
{
   if(dk(X))
         A(X);
   B(X);
   F(f(X));
   C(X);
   F(g(x));
}
```

Giải thuật khử đệ quy dùng Stack trong trường hợp này như sau:

```
F(X)
{
   taoStack (S);
   Push(S, (X,1));

   do
   {
         while(dk(X)==flase)
         {
               B(X);
               Push(S, (X,2));
               X = f(X);
         }
         A(X);
         Pop(S,(X,k));
         if(n!=1)
      {
         C(X);
         X=g(X);
      }
   }while(n>1);
}
```

Ví dụ 3.15:

Xét giải thuật đệ quy của bài toán tháp Hà Nội.

```
void HaNoiTower(int n, char A, char B, char C)
{
         if(n>0)
         {
               HaNoiTower(n-1, A, C, B);
               Move(A, C);
               HaNoiTower(n-1, B, A, C);
         }
         return; //nếu n==0 thì thực hiện 0 thao tác
   }
```

Trong đó:

- X là bộ (n,A, B,C).
- dk(X) là n<=0.
- A(X) và B(X) rỗng.
- C(X) là Move (A,C).
- f(X) là HaNoiTower(n-1, A, C, B).
- g(X) là HaNoiTower(n-1, B,A, C)

Khi đó giải thuật khử đệ quy dùng stack tương ứng như sau:

```
void HaNoiTower (int n, char A, char B, char C)
{
        taoStack(S) ;
        Push (S, (n,A,B,C,1)) ;
        do
        {
                while(n>0)
                {
                        Push(S,(n,A,B,C,2)) ;
                        n=n-1 ;
                        swap (B,C) ; //hoán đổi giá trị giữa B và C
                }
                Pop (S, (n,A,B,C,k)) ;
                if(k !=1)
                {
                        Move(A,C) ;
                        n=n-1 ;
                        Swap(A,B) ;
                }
        }while(k>1) ;
}
```

BÀI TẬP CHƯƠNG 3

Bài 1. Viết hàm tính các biểu thức S(n) theo 2 cách đệ quy và khử đệ quy (nếu có thể), với n là số nguyên dương nhập từ bàn phím :

1. S(n) = 1 + 2 + 3 + … + n.

2. S(n) = $\sqrt{2 + \sqrt{2 + \cdots + \sqrt{2 + \sqrt{2}}}}$ có n dấu căn.

3. $S(n) = \frac{1}{2} + \frac{2}{3} + \cdots + \frac{n}{n+1}$

4. $S(n) = 1 + \frac{1}{3} + \frac{1}{5} + \cdots + \frac{1}{2n+1}$

5. S(n) = 1.2 + 2.3+ 3.4 + 4.5 +…..+ n.(n+1)

6. $S(n) = \frac{1}{1.2.3} + \frac{1}{2.3.4} + \frac{1}{3.4.5} + ... + \frac{1}{n.(n+1).(n+2)}$

7. $S(n) = 1^2 + 2^2 + \cdots + n^2$

8. $S(n) = 1 + (1 + 2) + (1 + 2 + 3) + \cdots + (1 + 2 + 3 + \cdots + n)$

9. $S(n) = -\frac{1+2}{2!} + \frac{3+4}{4!} - \frac{5+6}{6!} \ldots + (-1)^n \frac{(2n-1)+(2n)}{(2n)!}$

10. $S(n) = \frac{1.2!}{2 + \sqrt{3}} + \frac{2.3!}{3 + \sqrt{4}} + \frac{3.4!}{4 + \sqrt{5}} + \cdots + \frac{n.(n+1)!}{(n+1) + \sqrt{(n+2)}}$

11. $S(n) = \frac{1+\sqrt{1+2}}{2+\sqrt{3!}} + \frac{2+\sqrt{2+3}}{3+\sqrt{4!}} + \frac{3+\sqrt{3+4}}{4+\sqrt{5!}} + \cdots + \frac{n+\sqrt{n+n+1}}{(n+1)+\sqrt{(n+2)!}}$

Bài 2. Viết hàm tính m^n với m là số nguyên và n là số nguyên dương nhập từ bàn phím.

Bài 3. Viết hàm tìm ước chung lớp nhất của 2 số nguyên dương a, b.

Gợi ý : Nếu a>b thì UCLN(a,b) = UCLN(b,a-b), ngược lại UCLN(a,b) = UCLN(a, b-a).

Bài 4. Viết hàm tìm giá trị phần tử thứ n của cấp số cộng có hạng đầu là a, công sai là r :

$$U_n = \begin{cases} a, nếu\ n = 1 \\ U_{n-1} + r, nếu\ n > 1 \end{cases}$$

Bài 5. Viết hàm tìm giá trị phần tử thứ n của cấp số nhân có hạng đầu là a, công bội là q :

$$U_n = \begin{cases} a, nếu\ n = 1 \\ qU_{n-1}, nếu\ n > 1 \end{cases}$$

Bài 6. Viết hàm tính biểu thức U(n) sau đây, với n là số nguyên dương nhập từ bàn phím :

$$U_n = \begin{cases} n, với\ n < 6 \\ U_{n-5} + U_{n-4} + U_{n-3} + U_{n-2} + U_{n-1}\ với\ n \geq 6 \end{cases}$$

Bài 7. Cho dãy số A_n theo các công thức quy nạp như sau, hãy viết chương trình tính số hạng thứ n, với n là số nguyên dương :

1. $A_0 = 1$; $A_1=0$; $A_2=-1$; $A_n = 2A_{n-1} – 3A_{n-2} – A_{n-3.}$

2. $A_1 = 1$; $A_2=2$; $A_3=3$; $A_{n+3} = 2A_{n+2} + A_{n+1} - 3A_n$

Viết chương trình tính số hạng thứ n.

Bài 8. Với $n \geq 1$, n là số nguyên dương, biết rằng f(n) được tính theo công thức đệ quy như sau : f(1) = 1 ; f(2n) =2f(n) ; f(2n+1) = 2f(n) + 3f(n+1).

1. Tính f(5).

2. Viết chương trình tính f(n).

Bài 10. Cho mảng 1 chiều a đã có thứ tự chứa n số nguyên, viết hàm tìm kiếm số x trên a theo thuật toán tìm kiếm nhị phân bằng kỹ thuật đệ quy.

Bài 11. Viết hàm xuất dãy có n số Fibonacci, biết rằng số Fibonacci là số có dạng :

$$F(n) = \begin{cases} 1, với\ i\ n \leq 2 \\ F(n-1) + F(n-2) \end{cases}$$

Ví dụ : nhập n=10→ dãy số Fibonacci có 10 số là : 1 1 2 3 5 8 13 21 34 55

Bài 12. Viết hàm tìm số Fibonacci lớn nhất nhưng nhỏ hơn số nguyên n cho trước theo 2 cách đệ quy và khử đệ quy.

Ví dụ : nhập n=15 → số Fibonacci lớn nhất nhỏ hơn 15 là 13.

Bài 13. Viết hàm tính số hạng thứ n của 2 dãy sau :

$x_0 = 1$, $y_0 = 0$,

$x_n = x_{n-1} + y_{n-1}$ với mọi n>0

$y_n = 3x_{n-1} + 2y_{n-1}$ với mọi n>n

Bài 14. Dãy An được cho như sau : $A_1=1$; $A_n = n(A_1+A_2+ … + A_{n-1})$.

Viết hàm tính A_n sử dụng kỹ thuật đệ quy.

Bài 15. Với mỗi $n \geq 1$, số Yn được tính như sau :

$Y_1=1$; $Y_2=2$; $Y_3=3$; $Y_n = Y_{n-1} + 2Y_{n-2} + 3Y_{n-3}$ nếu $n \geq 4$

Viết hàm tính Y_n bằng 2 cách đệ quy và khử đệ quy.

Bài 16. Với mỗi $n \geq 1$, số X_n được tính như sau :

$X_1=1$; $X_2=1$; $X_n = X_{n-1} + (n-1)X_{n-2}$ với $n \geq 3$

Viết hàm tính X_n bằng 2 cách đệ quy và khử đệ quy.

Bài 17. Cho dãy số xn được định nghĩa như sau :

$x_0 = 1$; $x_1 = 2$; $x_n = nx_0 + (n-1)x_1 + …+x_{n-1}$

Viết hàm đệ quy tính x_n với $n \geq 0$.

Bài 18. Dãy An được cho như sau :

$A_1=1$

$A_{2n} = n + A_n + 2$

$A_{2n+1} = n^2 + A_n.A_{n+1} + 1$

Viết hàm tìm tất cả các dãy nhị phân chiều dài n, với $n \geq 0$.

Bài 19. Bài toán 8 quân hậu : đặt 8 quân hậu lên bàn cờ vua sao cho chúng không khống chế lẫn nhau. Viết chương trình tìm tất cả các lời giải.

Bài 20. Bài toán con mã đi tuần : cho bàn cờ kích thước 8×8, hãy di chuyển quân mã trên khắp bàn cờ sao cho mỗi ô chỉ đi đúng 1 lần. Viết chương trình tìm tất cả cách đi của quân mã.

CHƯƠNG 4. KỸ THUẬT XỬ LÝ CHUỖI

4.1 Một số khái niệm

4.1.1 Chuỗi kí tự

Chuỗi kí tự, hay còn gọi là xâu kí tự, là một dãy các kí tự viết liền nhau. Trong đó, các kí tự được lấy từ bảng chữ cái ASCII. Chuỗi kí tự được hiểu là một mảng 1 chiều chứa các kí tự.

Cách khai báo chuỗi kí tự như sau:

```
char s[100];
```

hoặc

```
char *s = new char[100];
```

Ví dụ trên là khai báo một chuỗi kí tự s có độ dài tối đa 100 kí tự, trong đó chuỗi s có tối đa 99 bytes tương ứng 99 kí tự có ý nghĩa trong chuỗi, và byte cuối cùng lưu kí tự kết thúc chuỗi là '\0'. Kí hiệu '\0' là kí tự bắt buộc dùng để kết thúc một chuỗi.

Hằng xâu kí tự được ghi bằng cặp dấu nháy kép. Ví dụ, "Hello". Nếu giữa cặp dấu nháy kép không ghi kí tự nào thì ta có chuỗi rỗng và độ dài chuỗi rỗng bằng 0.

4.1.2 Nhập/ xuất chuỗi kí tự

Trong ngôn ngữ lập trình C, ta có thể sử dụng hàm scanf với kí tự định dạng là %s để nhập một chuỗi kí tự do người dùng nhập vào từ bàn phím vào chương trình.

```
char str[100];
scanf("%s", &str);
```

Nhược điểm của hàm scanf khi nhập nội dung chuỗi kí tự có khoảng trắng thì kết quả lưu trong chuỗi không đúng như người dùng mong muốn. Khi nhập chuỗi có chứa kí tự khoảng trằng thì biên kiểu chuỗi chỉ lưu được phần đầu chuỗi đến khi gặp khoảng trắng đầu tiên, phần còn lại được lưu vào vùng nhớ đệm để gán cho biến kiểu chuỗi tiếp sau khi gặp lệnh scanf dịnh dạng chuỗi lần kế tiếp.

Thông thường, để nhập một chuỗi ký tự từ bàn phím, ta sử dụng hàm gets().

Cú pháp: gets(<Biến chuỗi>)

Ví dụ 4.1:

```
char str[100];
gets(str);
```

Để xuất một chuỗi (biểu thức chuỗi) lên màn hình, ta sử dụng hàm puts().

Cú pháp: puts(<Biểu thức chuỗi>)

Ví dụ 4.2:

```
puts(str);
```

Một chương trình thực thi sử dụng nhiều biến lưu trữ dữ liệu. Trong khi đó, vùng nhớ chương trình thực thi thì hạn chế, do đó, người dùng thường lưu trữ dữ liệu trên file text để hỗ trợ cho chương trình thực thi tốt.

Cho f là input file dạng text thì dòng lệnh **f >> s** đọc dữ liệu vào đối tượng s đến khi *gặp dấu cách*.

Ví dụ 4.3:

Trong file input.txt chứa thông tin sau:

35 4 5 11

Đoạn lệnh sau đây sẽ gọi 4 lần dòng lệnh f>>s để thực hiện chức năng đọc thông tin trong file input.txt và in nội dung đọc được trong file ra màn hình.

```
ifstream f("input.txt");
char s[100];
for(int i=0; i<4; i++)
{
     f>>s;
     cout<<"\t"<<s;
}
```

Muốn đọc đầy đủ một dòng dữ liệu chứa cả dấu cách từ input file f vào một biến mảng kí tự s ta có thể dùng phương thức getline như ví dụ sau đây

```
char s[1001];
f.getline(s,1000,'\n');
```

Phương thức này đọc một dòng tối đa 1000 kí tự vào biến s, và thay dấu kết dòng '\n' trong input file bằng dấu kết xâu '/0' trong C.

Ta có thể sử dụng phương thức g<<s cho phép ghi chuỗi s vào file dạng text, trong đó g là output file dạng text.

```
ofstream g("output.txt");
g<< " Hello!";
```

Sau khi kết thúc thao tác, ta dùng phương thức đóng file

```
f.close();
g.close();
```

4.1.3 Xâu con

Cho 1 xâu kí tự s. Nếu xóa khỏi s một số kí tự và dồn các kí tự còn lại cho kề nhau, ta sẽ thu được một *xâu con* của xâu kí tự s.

Ví dụ 4.4:

s = "Ky thuat lap trinh";

S1 = "Ky lap trinh", S2 = "Ky thuat", S3 = "thuat" là xâu con của xâu kí tự s.

Cho 2 xâu kí tự s1, s2. Một xâu kí tự s vừa là xâu con của s1, và vừa là xâu con của s2 thì s được gọi là xâu con chung của 2 chuỗi s1 và s2.

Xét x = "xaxxbxcxd", y = "ayybycdy", xâu con chung của x, y là "abcd", "abd", "acd",… và chiều dài của xâu con chung dài nhất là 4.

Thuật toán xác định chiều dài xâu con chung dài nhất.

Xét hàm 2 biến s(i,j) là đáp số khi giải bài toán với 2 tiền tố i:x và j:y. Ta có:

- s(0,0) = s(i,0) = s(0,j) = 0: một trong hai xâu là rỗng thì xâu con chung là rỗng nên chiều dài là 0;
- Nếu x[i] = y[j] thì s(i,j) = s(i–1,j–1) + 1;
- Nếu x[i] ≠ y[j] thì s(i,j) = Max { s(i–1,j), s(i,j–1) }.

Để cài đặt, trước hết ta có thể sử dụng mảng hai chiều v với qui ước v[i][j] = s(i,j). Sau đó ta cải tiến bằng cách sứ dụng 2 mảng một chiều a và b, trong đó a là mảng đã tính ở bước thứ i–1, b là mảng tính ở bước thứ i, tức là ta qui ước a = v[i–1] (dòng i–1 của ma trận v), b = v[i] (dòng i của ma trận v). Ta có, tại bước i, ta xét kí tự x[i], với mỗi j = 0..len(y)–1,

- Nếu x[i] = y[j] thì b[j] = a[j–1] + 1;
- Nếu x[i] ≠ y[j] thì b[j] = Max { a[j], b[j–1] }.

Sau khi đọc dữ liệu vào hai xâu x và y ta gọi hàm XauChung để xác định chiều dài tối đa của xâu con chung của x và y. a,b là các mảng nguyên 1 chiều.

4.2 Các thuật toán tìm kiếm chuỗi

Các thuật toán này đều có cùng ý nghĩa là kiểm tra một chuỗi P có nằm trong một văn bản T hay không, nếu có thì nằm ở vị trí nào, và xuất hiện bao nhiêu lần. Ví dụ, kiểm tra chuỗi "lập trình" có nằm trong nội dung của file văn bản KTLT.txt hay không, xuất hiện tại vị trí nào và xuất hiện bao nhiêu lần.

4.2.1 Thuật toán Brute Force

Thuật toán Brute Force thử kiểm tra tất cả các vị trí trên văn bản từ 0 cho đến kí tự cuối cùng trong văn bản. Sau mỗi lần thử thuật toán Brute Force dịch mẫu sang phải một ký tự cho đến khi kiểm tra hết văn bản.

Thuật toán Brute Force không cần giai đoạn tiền xử lý cũng như các mảng phụ cho quá trình tìm kiếm. Độ phức tạp tính toán của thuật toán này là O(N*M).

Để mô phỏng quá trình tìm kiếm, ta thu nhỏ văn bản T thành chuỗi T. Ý tưởng của thuật toán này là so sánh từng kí tự trong chuỗi P với từng kí tự trong đoạn con của T. Bắt đầu từng kí tự trong P so sánh cho đến hết chuỗi P, trong quá trình so sánh, nếu thấy có sự sai khác giữa P và 1 đoạn con của T thì bắt đầu lại từ kí tự đầu tiên của P, và xét kí tự tiếp sau kí tự của lần so sánh trước trong T.

Thuật toán

```
//Nhập chuỗi chính T
Nhập chuỗi cần xét P
if ((  len (P) = 0) or ( len (T) = 0 ) or (  len (P) > len (T) )
 // len : chiều dài chuỗi
   P không xuất hiện trong T
else
   Tính vị trí dừng STOP =  len(T) – len(P)
   Vị trí bắt đầu so sánh START = 0
// ta cần vị trí dừng vì ra khỏi STOP
//chiều dài P lớn hơn chiều dài đoạn T còn lại thì dĩ nhiên P không thuộc T
```

So sánh các ký tự giữa P và T bắt đầu từ START

IF (các ký tự đều giống nhau)

P có xuất hiện trong T

ELSE

Tăng START lên 1

Dừng khi P xuất hiện trong T hoặc START > STOP

Ví dụ 4.5:

T = “I LIKE COMPUTER” n = len(T) = 14

P = “LIKE” m = len(P) = 4

start = 0

stop = n-m = 10

Bước 1: I LIKE COMPUTER i = start = 0

LIKE j = 0

P[j] = 'L' != T[i+j] = 'I'

----> tăng i lên 1 (ký tự tiếp theo trong T)

j = 0;(trở về đầu chuỗi P so sánh lại từ đầu P hay P dịch sang phải 1 ký tự.

Bước 2: I LIKE COMPUTER i = 1

LIKE j = 0

p[j] = 'L' != T[i] = ' '

---> tăng i lên một, j = 0

Bước 3: I LIKE COMPUTER i = 2

LIKE j = 0

p[j] == T[i+j] ='L'

----> tăng j lên một,

không tăng i vì ta đang xét T[i+j].

(chỉ khi nào có sự sai khác mới tăng i lên một)

Bước 4: I LIKE COMPUTER i = 2

LIKE j = 1

p[j] == T[i+j] = 'I'

-----> tăng j lên một

Bước 5: I LIKE COMPUTER i = 2

LIKE j = 2

p[j] == T[i+j] = 'K'

-----> tăng j lên một

Bước 6: I LIKE COMPUTER i = 2

LIKE j = 3

p[j] == T[i+j] = 'I'

-----> tăng j lên một -----> j = 4 = m : hết chuỗi P ------> P có xuất hiện trong T

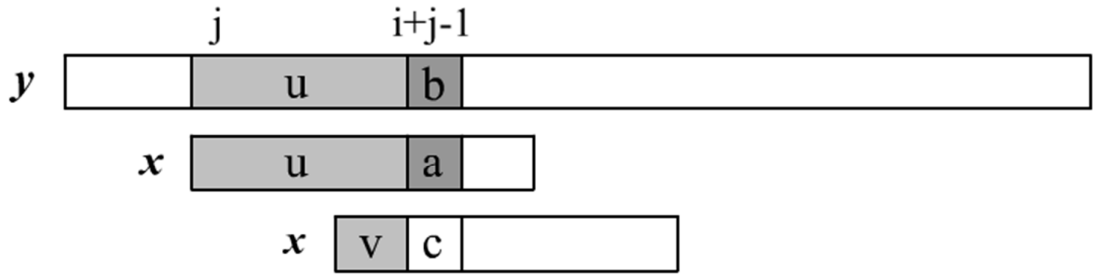
Kết quả: Xuất ra vị trí i = 2 là vị trí xuất hiện đầu tiên của P trong T

4.2.2 Thuật tóan Knuth – Morris – Pratt

Thuật toán Knuth-Morris-Pratt là thuật toán có độ phức tạp tuyến tính đầu tiên được phát hiện ra, nó dựa trên thuật toán brute force với ý tưởng lợi dụng lại những thông tin của lần thử trước cho lần sau. Trong thuật toán brute force vì chỉ dịch cửa sổ đi một ký tự nên có đến m-1 ký tự của cửa sổ mới là những ký tự của cửa sổ vừa xét. Trong đó có thể có rất nhiều ký tự đã được so sánh giống với mẫu và bây giờ lại nằm trên cửa sổ mới nhưng được dịch đi về vị trí so sánh với mẫu. Việc xử lý những ký tự này có thể được tính toán trước rồi lưu lại kết quả. Nhờ đó lần thử sau có thể dịch đi được nhiều hơn một ký tự, và giảm số ký tự phải so sánh lại.

Xét lần thử tại vị trí j, khi đó cửa sổ đang xét bao gồm các ký tự *y[j…j+m-1]* giả sử sự khác biệt đầu tiên xảy ra giữa hai ký tự *x[i]* và *y[j+i-1]*.

Khi đó *x[1…i]=y[j…i+j-1]=u* và *a=x[i]¹y[i+j]=b*. Với trường hợp này, dịch cửa sổ phải thỏa mãn *v* là phần đầu của xâu x khớp với phần đuôi của xâu u trên văn bản. Hơn nữa ký tự *c* ở ngay sau *v* trên mẫu phải khác với ký tự *a*. Trong những đoạn như *v* thoả mãn các tính chất trên ta chỉ quan tâm đến đoạn có độ dài lớn nhất.



Dịch cửa sổ sao cho *v* phải khớp với *u* và *c ≠ a*

Dịch cửa sổ sao cho *v* phải khớp với *u* và *c ¹ a*

Thuật toán Knuth-Morris-Pratt sử dụng mảng Next[i] để lưu trữ độ dài lớn nhất của xâu v trong trường hợp xâu *u=x[1…i-1]*. Mảng này có thể tính trước với chi phí về thời gian là ***O(m)*** (việc tính mảng Next thực chất là một bài toán qui hoạch động một chiều).

Thuật toán Knuth-Morris-Pratt có chi phí về thời gian là ***O(m+n)*** với nhiều nhất là 2n-1 lần số lần so sánh ký tự trong quá trình tìm kiếm.

Ví dụ

Để minh họa chi tiết thuật toán, chúng ta sẽ tìm hiểu từng quá trình thực hiện của thuật toán. Ở mỗi thời điểm, thuật toán luôn được xác định bằng hai biến kiểu nguyên, m và i, được định nghĩa lần lượt là vị trí tương ứng trên S bắt đầu cho một phép so sánh với W, và *chỉ số* trên W xác định kí tự đang được so sánh. Khi bắt đầu, thuật toán được xác định như sau:

m: 0

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

i: 0

Chúng ta tiến hành so sánh các kí tự của `W` tương ứng với các kí tự của `S`, di chuyển lần lượt sang các chữ cái tiếp theo nếu chúng giống nhau. `S[0]` và `W[0]` đều là ‘A’. Ta tăng i :

m: 0

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

i: _1

`S[1]` và `W[1]` đều là ‘B’. Ta tiếp tục tăng i :

m: 0

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

i: __2

`S[2]` và `W[2]` đều là ‘C’. Ta tăng i lên 3 :

m: 0

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

i: ___3

Nhưng, trong bước thứ tư, ta thấy `S[3]` là một khoảng trống trong khi `W[3] = 'D'`, không phù hợp. Thay vì tiếp tục so sánh lại ở vị trí `S[1]`, ta nhận thấy rằng không có kí tự `'A'` xuất hiện trong khoảng từ vị trí 0 đến vị trí 3 trên xâu `S` ngoài trừ vị trí 0; do đó, nhờ vào quá trình so sánh các kí tự trước đó, chúng ta thấy rằng không có khả năng tìm thấy xâu dù có so sánh lại. Vì vậy, chúng ta di chuyển đến kí tự tiếp theo, gán `m = 4` và `i = 0`.

m: ____4

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

i: 0

Tiếp tục quá trình so sánh như trên, ta xác định được xâu chung `"ABCDAB"`, với `W[6]` (`S[10]`), ta lại thấy không phù hợp. Nhưng từ kết quả của quá trình so sánh trước, ta đã duyệt qua `"AB"`, có khả năng sẽ là khởi đầu cho một đoạn xâu khớp, vì vậy ta bắt đầu so sanh từ vị trí này. Như chúng ta đã thấy các kí tự này đã trùng khớp với hau kí tự trong phép so khớp trước, chúng ta không cần kiểm tra lại chúng một lần nữa; ta bắt đầu với `m = 8`, `i = 2` và tiếp tục quá trình so khớp.

m: _____8

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

i: __2

Quá trình so khớp ngay lập tức thất bại, nhưng trong W không xuất hiện kí tự ' ',vì vậy, ta tăng m lên 11, và gán i = 0.

m: _____11

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

i: 0

Một lần nữa, hai xâu trùng khớp đoạn kí tự "ABCDAB" nhưng ở kí tự tiếp theo, 'C', không trùng với 'D' trong W. Giống như trước, ta gán m = 15, và gán i = 2, và tiếp tục so sánh.

m: _____15

S: ABC ABCDAB ABCDABCDABDE

W: ABCDABD

i: __2

Lần này, chúng ta đã tìm được khớp tương ứng với vị trí bắt đầu là S[15].

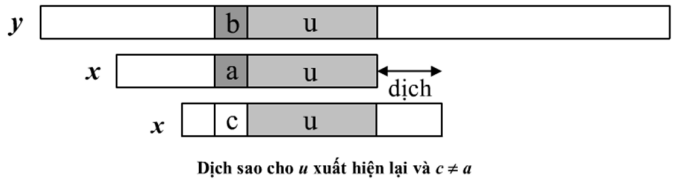
4.2.3 Thuật tóan Boyer Moore

Thuật toán Boyer Moore là thuật toán có tìm kiếm chuỗi rất có hiệu quả trong thực tiễn, các dạng khác nhau của thuật toán này thường được cài đặt trong các chương trình soạn thảo văn bản.

Khác với thuật toán Knuth-Morris-Pratt, thuật toán Boyer-Moore kiểm tra các ký tự của mẫu từ phải sang trái và khi phát hiện sự khác nhau đầu tiên thuật toán sẽ tiến hành dịch cửa sổ đi. Trong thuật toán này có hai cách dịch của sổ:

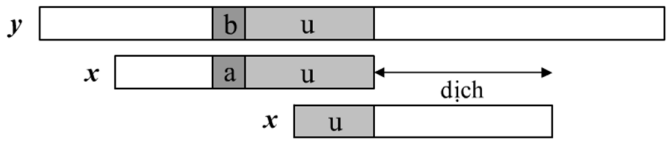
Cách thứ 1: gần giống như cách dịch trong thuật toán KMP, dịch sao cho những phần đã so sánh trong lần trước khớp với những phần giống nó trong lần sau.

Trong lần thử tại vị trí *j,* khi so sánh đến ký tự *i* trên mẫu thì phát hiện ra sự khác nhau, lúc đó $x[i+1…m]=y[i+j...j+m-1]=u$ và $a=x[i]^1y[i+j-1]=b$ khi đó thuật toán sẽ dịch cửa sổ sao cho đoạn $u=y[i+j…j+m-1]$ giống với một đoạn mới trên mẫu (trong các phép dịch ta chọn phép dịch nhỏ nhất)



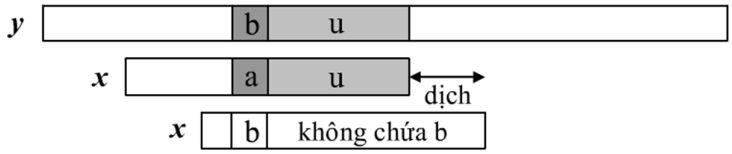
Dịch sao cho *u* xuất hiện lại và $c \neq a$

Nếu không có một đoạn nguyên vẹn của u xuất hiện lại trong x, ta sẽ chọn sao cho phần đôi dài nhất của u xuất hiện trở lại ở đầu mẫu.



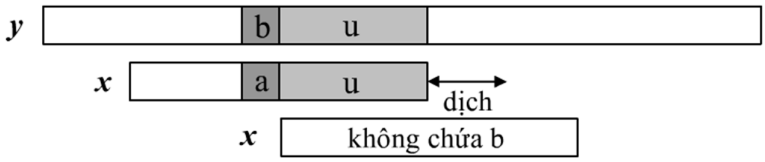
Dịch để một phần đôi của u xuất hiện lại trên x

Cách thứ 2: Coi ký tự đầu tiên không khớp trên văn bản là *b=y[i+j-1]* ta sẽ dịch sao cho có một ký tự giống *b* trên xâu mẫu khớp vào vị trí đó (nếu có nhiều vị trí xuất hiện *b* trên xâu mẫu ta chọn vị trí phải nhất)



Dịch để ký tự b ăn khớp với văn bản.

Nếu không có ký tự b nào xuất hiện trên mẫu ta sẽ dịch cửa sổ sao cho ký tự trái nhất của cửa sổ vào vị trí ngay sau ký tự *y[i+j-1]=b* để đảm bảo sự ăn khớp



Dịch khi b không xuất hiện trong *x*

Trong hai cách dịch thuật toán sẽ chọn cách dịch có lợi nhất.

Trong cài đặt ta dùng mảng bmGs để lưu cách dịch 1, mảng bmBc để lưu phép dịch thứ 2(ký tự không khớp). Việc tính toán mảng bmBc thực sự không có gì nhiều để bàn. Nhưng việc tính trước mảng bmGs khá phức tạp, ta không tính trực tiếp mảng này mà tính gián tiếp thông qua mảng suff. Có suff[i]=max{k | x[i-k+1…i]=x[m-k+1…m]}

Các mảng bmGs và bmBc có thể được tính toán trước trong thời gian tỉ lệ với ***O(m+d)***. Thời gian tìm kiếm (độ phức tạp tính toán) của thuật toán Boyer-Moore là ***O(m*n)***. Tuy nhiên với những bản chữ cái lớn thuật toán thực hiện rất nhanh. Trong trường hợp tốt chi phí thuật toán có thể xuống đến ***O(n/m)*** là chi phí thấp nhất của các thuật toán tìm kiếm hiện đại có thể đạt được.

BÀI TẬP CHƯƠNG 4

Bài 1. Cho file input.txt chứa nội dung sau:

Môn học: Kỹ thuật lập trình nâng cao

Số tiết: 30

Viết chương trình đọc nội dung trong file input.txt ra màn hình theo đúng định dạng trong file.

Bài 2. Cho file input.txt chứa nội dung sau:

6

35 2 6 4 12 9

Viết chương trình đọc nội dung trong file input.txt lưu vào mảng 1 chiều, sau đó in ra màn hình tổng các phần tử trong mảng.

Bài 3. Cho file input.txt chứa nội dung sau:

Số phần tử là: 5

Giá trị mảng la: 4 2 7 13 9

Viết chương trình đọc nội dung trong file input lưu vào biến mảng 1 chiều, sắp xếp mảng 1 chiều tăng dần, sau đó lưu kết quả mảng đã sắp xếp vào file output.txt.

Bài 4. Cho file input chứa nội dung sau:

Số dòng: 4

Số cột: 5

3 6 8 0 -5

5 12 7 21 -45

2 6 19 3 4

1 6 -7 3 9

Viết chương trình đọc nội dung trong file input lưu vào biến mảng 2 chiều, tìm dòng có tổng lớn nhất, sau đó lưu kết quả vào file output.txt.

Bài 5. Cho file input.txt chứa nội dung sau:

Số phần tử: 4

3 6 2 9

4 7 9 1

5 2 6 8

4 2 7 3

Viết chương trình đọc nội dung trong file input, đếm số phần tử chẵn trên biên của ma trận, sau đó in kết quả ra màn hình.

Bài 6. Cho file input.txt chứa nội dung sau:

Số phần tử: 3

Ma trận A

3	6	3
4	3	1
2	5	2

Ma trận B

1	4	9
8	3	7
2	5	6

Viết chương trình đọc nội dung trong file input, tính ma trận tổng S = A+B, ma trận tích P = AxB, sau đó lưu 2 ma trận kết quả vào file output.txt

Bài 7. Viết chương trình cài đặt thuật toán tìm chiều dài xâu con chung dài nhất trong 2 chuỗi ký tự cho trước.

Bài 8. Viết chương trình cài đặt thuật toán tìm kiếm chuỗi Brute Force.

Bài 9. Viết chương trình cài đặt thuật toán tìm kiếm chuỗi Knuth – Morris – Pratt

Bài 10. Viết chương trình cài đặt thuật toán tìm kiếm chuỗi Boyer – Moore

CHƯƠNG 5. THIẾT KẾ THUẬT TOÁN

5.1 Kỹ thuật chia để trị - Divide to Conquer

5.1.1 Khái niệm

Chia để trị là một trong những kỹ thuật phổ biến được sử dụng để giải bài toán bằng cách chia bài toán gốc thành một hoặc nhiều bài toán đồng dạng có kích thước nhỏ hơn, rồi giải lần lượt từng bài toán nhỏ một cách độc lập. Lời giải của bài toán gốc chính là sự kết hợp lời giải của những bài toán con. Từ lâu, đã có rất nhiều thuật giải kinh điển dựa trên phương pháp này như thuật giải tìm kiếm nhị phân (Binary Search), thuật giải sắp xếp nhanh (Quick Sort), thuật giải sắp xếp trộn (Merge Sort)…

Các bước thực hiện kỹ thuật chia để trị :

- Divide : chia bài toán ban đầu thành một số bài toán con.
- Conquer : giải quyết các bài toán con. Chúng ta có thể giải quyết các bài toán con bằng đệ quy hoặc kích thước bài toán đủ nhỏ thì giải trực tiếp.
- Combine : kết hợp lời giải của các bài toán con thành lời giải của bài toán ban đầu. Trong một số bài toán thì có thể không cần đến bước này.

Mã giả của giải thuật chia để trị như sau :

```
Solve (n)
{
        if (n đủ nhỏ để có thể giải được trực tiếp bài toán)
        {
                Giải bài toán → Kết quả.
                return Kết quả ;
        }
        else
        {
                Chia bài toán thành các bài toán con kích thước n1, n2, …
                Kết quả 1= Solve (n1) ; //giải bài toán con 1.
                Kết quả 2= Solve (n2) ; //giải bài toán con 2.
                …
                //Tổng hợp các kết quả Kết quả 1, kết quả 2→ kết quả.
                return kết quả ;
        }
}
```

Lưu ý :

- Bước phân chia càng đơn giản thì bước tổng hợp càng phức tạp và ngược lại.

- Đối với một số bài toán, việc tổng hợp lời giải của các bài toán con là không cần thiết vì nó được bao hàm trong bước phân chia bài toán. Do đó khi giải xong các bài toán con thì bài toán ban đầu cũng đã được giải xong.

5.1.2 Một số bài toán minh họa

Để minh họa cho việc sử dụng giải thuật chia để trị, ta áp dụng với một số thuật giải sau: tìm kiếm nhị phân (Binary Search), sắp xếp theo trộn phần tử (Merge Sort), sắp xếp nhanh (Quick Sort).

5.1.2.1 Bài toán tìm kiếm nhị phân

Mô tả bài toán: Bài toán tìm kiếm gồm dữ liệu đầu vào là một mảng n phần tử đã có thứ tự, một khóa *key* kèm theo để so sánh giữa các phần tử trong mảng. Kết quả của bài toán là có phần tử nào trong mảng bằng với *key* không ?

Xây dựng thuật giải tìm kiếm nhị phân từ thuật giải chia để trị tổng quát.

Ý tưởng : Giả sử ta có mảng A có thứ tự tăng dần. Khi đó $A_i < A_j$ với i<j.

- Divide: xác định phần tử giữa mảng là A_{mid}. Nếu phần tử cần tìm bằng phần tử này thì trả về vị trí tìm được và kết thúc. Nếu phần tử cần tìm nhỏ hơn A_{mid} thì ta chỉ cần tìm trong dãy con bên trái A_{mid}. Nếu phần tử cần tìm lớn hơn phần tử này thì ta chỉ cần tìm trong dãy con bên phải A_{mid}.
- Conquer: Tiếp tục tìm kiếm trong các dãy con đến khi tìm thấy khóa key hoặc đến khi hết dãy khi không có key trong dãy.
- Combine: Không cần trong trường hợp này.

Mã giả giải thuật

```
Binary_Search( A, n, key)
{
     left = 0; // vị trí phần tử đầu tiên trong mảng
     right = n-1; // vị trí phần tử cuối cùng trong mảng
     while (left <= right)
     {
          mid = (left + right)/2;       //vị trí giữa mảng
          if (A[mid] == key)
               return mid;
          if (key < A[mid])
             right = mid – 1 ;
          else
             left = mid + 1;
     }
     return -1; // không tìm thấy key trong mảng nên trả về vị
trí -1.
}
```

5.1.2.2 Bài toán sắp xếp theo giải thuật Merge Sort

Mô tả bài toán: Bài toán sắp xếp gồm dữ liệu đầu vào là một mảng các phần tử. Kết quả của bài toán là mảng đã được xếp theo thứ tự (tăng hoặc giảm).

Xây dựng thuật giải sắp xếp Merge Sort từ thuật giải chia để trị tổng quát.

Ý tưởng : Giả sử ta có mảng A có n phần tử.

- Divide: Chia dãy n phần tử cần sắp xếp thành 2 dãy con, mỗi dãy có n/2 phần tử.
- Conquer: Sắp xếp các dãy con bằng cách gọi đệ quy Merge Sort. Dãy con chỉ có 1 phần tử thì mặc nhiên có thứ tự, không cần sắp xếp.
- Combine: Trộn 2 dãy con đã sắp xếp để tạo thành dãy ban đầu có thứ tự.

Mã giả giải thuật

```
Merge_Sort (A, 0, n)
{
        left = 0; // vị trí phần tử đầu tiên của dãy.
        right = n-1; // ví trị phần tử cuối cùng của dãy
        if (left < right)
        {
         mid = (left + right)/2;        //vị trí phần tử ở giữa dãy
         Merge_Sort (A, left, mid); // Gọi hàm Merge_Sort cho nửa dãy con đầu
         Merge_Sort (A, mid+1, right); // Gọi hàm Merge_Sort cho nửa dãy con cuối
         Merge (A, left, mid, right); //Hàm trộn 2 dãy con có thứ tự thành dãy ban đầu có thứ tự
        }
}
//-------------------------------------------------------------------------------------------------
Merge (A, left, mid, right)
{
     n1 = mid - left + 1; //độ dài nửa dãy đầu của A.
     n2 = right - mid; // độ dài nửa dãy sau của A
     L[], R[]; //L là dãy chứa nửa dãy đầu của A; R là dãy chứa nửa dãy sau của A.
     for i = 0  to n1-1
       L[i] = A[left+i]; //chép nửa dãy đầu của A vào L.
     for j = 0 to n2-1
   R[j] = A[mid + j + 1] //chép nửa dãy sau của A vào R.
     i=0
     j=0
```

```
     for k = left to right          // L và R lại vào A sao
cho A có thứ tự tăng dần.
          if (L[i] <= R[j])
          {
               A[k] = L[i]
               i = i+1
          }
          else
          {
               A[k] = R[j]
               j = j + 1
          }
}
```

5.1.2.3. Bài toán nhân các số nguyên lớn

Mô tả bài toán: Trong các ngôn ngữ lập trình đều có kiểu dữ liệu số nguyên (chẳng hạn kiểu integer trong Pascal, Int trong C…), nhưng nhìn chung các kiểu này đều có miền giá trị hạn chế (chẳng hạn từ -32768 đến 32767) nên khi có một ứng dụng trên số nguyên lớn (hàng chục, hàng trăm chữ số) thì kiểu số nguyên định sẵn không đáp ứng được. Trong trường hợp đó, người lập trình phải tìm một cấu trúc dữ liệu thích hợp để biểu diễn cho một số nguyên, chẳng hạn ta có thể dùng một chuỗi kí tự để biểu diễn cho một số nguyên, trong đó mỗi kí tự lưu trữ một chữ số. Để thao tác được trên các số nguyên được biểu diễn bởi một cấu trúc mới, người lập trình phải xây dựng các phép toán cho số nguyên như phép cộng, phép trừ, phép nhân… Bài toán sau đây sẽ đề cập đến bài toán nhân hai số nguyên lớn.

Xét bài toán nhân 2 số nguyên lớn X và Y, mỗi số có n chữ số.

Theo cách nhân thông thường mà ta đã được học ở phổ thông thì phép nhân được thực hiện như sau: nhân từng chữ số của Y với X (kết quả được dịch trái 1 vị trí sau mỗi lần nhân) sau đó cộng các kết quả lại.

Ví dụ 5.1 : X = 2357, Y = 4891, ta đặt tính nhân như sau:

```
          2357
         ×
          4891
        ------------
          2357
     + 21213
     18856
     9428
-------------------------
    11528087
```

Việc nhân từng chữ số của X và Y tốn n^2 phép nhân (vì X và Y có n chữ số). Nếu phép nhân một chữ số của X cho một chữ số của Y tốn O(1) thời gian, thì độ phức tạp giải thuật của giải thuật nhân X và Y này là $O(n^2)$.

Xây dựng thuật giải nhân số nguyên lớn từ thuật giải chia để trị tổng quát.

Ý tưởng :

Áp dụng kĩ thuật "chia để trị" vào phép nhân các số nguyên lớn, ta chia mỗi số nguyên lớn X và Y thành các số nguyên lớn có n/2 chữ số. Để việc phân tích giải thuật đơn giản, ta giả sử n là luỹ thừa của 2, còn về khía cạnh lập trình, vì máy xử lý nên ta vẫn có thể viết chương trình với n bất kì.

- Biểu diễn X và Y dưới dạng sau: X = A.10n/2 + B ; Y = C.10n/2 + D.
- Trong đó A, B, C, D là các số có n/2 chữ số. Chẳng hạn với X = 7853 thì A = 78 và B=53 vì $78*10^2 + 53 = 7853 = X$
- Do đó, với cách biểu diễn trên thì $XY = A.C.10^n + (A.D + B.C)10^{n/2} + B.D$ (*)

Khi đó thay vì nhân trực tiếp 2 số có n chữ số, ta phân tích bài toán ban đầu thành một số bài toán nhân 2 số có n/2 chữ số. Sau đó, ta kết hợp các kết quả trung gian theo cách phân tích và công thức (*). Việc phân chia này sẽ dẫn đến các bài toán nhân 2 số có 1 chữ số. Đây là bài toán cơ sở. Tóm lại các bước giải thuật chia để trị cho bài toán trên như sau:

- Divide: Chia bài toán nhân 2 số nguyên lớn X, Y có n chữ số thành các bài toán nhân các số có n/2 chữ số.
- Conquer: tiếp tục chia các bài toán nhân sao cho đưa về các bài toán nhân 2 số có 1 chữ số.
- Combine: Tổng hợp các kết quả trung gian theo công thức (*).

Mã giả giải thuật : nhân 2 số nguyên lớn X, Y có n chữ số.

```
Big_Number_Multi (Big_Int X, Big_Int Y, int n)
{
     Big_Int m1, m2, m3, A, B, C, D;
     int s; //lưu dấu của tích XY
s = sign(X) * sign(Y); //sign(X) trả về 1 nếu X dương; -1 là
âm; 0 là X = 0

     X = abs(X);
     Y = abs(Y);
     if (n==1) // X, Y có 1 chữ số
          return X*Y*s;
     else
     {
          A = left (X, n/2); // số có n/2 chữ số đầu của X.
```

```
    B = right(X, n/2); // số có n/2 chữ số cuối của X.
    C = left(Y, n/2);// số có n/2 chữ số đầu của Y
    D = right (Y, n/2);// số có n/2 chữ số cuối của Y

            m1 = Big_Number_Multi (A, C, n/2);
            m2 = Big_Number_Multi (A-B, D-C, n/2);
            m3 = Big_Number_Multi (B, D, n/2);
            return s* (m1*10^n + (m1 + m2 + m3)*10^(n/2) +m3);
    }
}
```

5.2 Kỹ thuật tham ăn – Greedy Technique

5.2.1 Giới thiệu bài toán tối ưu tổ hợp

Bài toán tối ưu tổ hợp có dạng tổng quát như sau:

- Cho hàm f(X) = ánh xạ trên một tập hữu hạn các phần tử D. Hàm f(X) được gọi là hàm mục tiêu.
- Mỗi phần tử X ∈ D có dạng X = (x1, x2, .. xn) được gọi là một phương án.
- Cần tìm một phương án X ∈D sao cho hàm f(X) đạt min (max). Phương án X như thế được gọi là phương án tối ưu.

Ta có thể tìm thấy phương án tối ưu bằng phương pháp "vét cạn" nghĩa là xét tất cả các phương án trong tập D (hữu hạn) để xác định phương án tốt nhất. Mặc dù tập hợp D là hữu hạn nhưng để tìm phương án tối ưu cho một bài toán kích thước n bằng phương pháp "vét cạn" ta có thể cần một thời gian mũ (nghĩa là thời gian tăng dạng số mũ theo giá trị n).

5.2.2 Nội dung kỹ thuật tham ăn.

Kỹ thuật tham ăn hay còn gọi là phương pháp tham lam. "Tham ăn" hiểu một cách đơn giản là: trong một bàn ăn có nhiều món ăn, món nào ngon nhất ta sẽ ăn trước và ăn cho hết món đó mới chuyển sang món thứ hai, tiếp tục ăn hết món thứ hai và chuyển sang món thứ ba,…

Kĩ thuật tham ăn thường được vận dụng để giải bài toán tối ưu tổ hợp bằng cách xây dựng một phương án X. Phương án X được xây dựng bằng cách lựa chọn từng thành phần Xi của X cho đến khi hoàn chỉnh (đủ n thành phần). Với mỗi Xi, ta sẽ chọn Xi tối ưu. Với cách này thì có thể ở bước cuối cùng ta không còn gì để chọn mà phải chấp nhận một giá trị cuối cùng còn lại.

Thực tế có nhiều bài toán chúng ta không thể tìm được đáp án tối ưu nhất mà chỉ có thể tìm được cách giải quyết tốt nhất có thể mà thôi, và kỹ thuật tham ăn là một trong những phương pháp được áp dụng phổ biến cho các loại bài toán tối ưu.

5.2.3 Một số bài toán minh họa

5.2.3.1 Bài toán đi đường của người giao hàng.

Một trong những bài toán nổi tiếng áp dụng kỹ thuật tham lam để tìm cách giải quyết đó là bài toán tìm đường đi của người giao hàng (TSP - Traveling Salesman Problem). Bài toán được mô tả như sau: *Có một người giao hàng cần đi giao hàng tại n thành phố. Xuất phát từ một thành phố nào đó, đi qua các thành phố khác để giao hàng và trở về thành phố ban đầu.*

Yêu cầu :

+ Mỗi thành phố chỉ đến một lần.
+ Khoảng cách từ một thành phố đến các thành phố khác là xác định được.
+ Giả thiết rằng mỗi thành phố đều có đường đi đến các thành phố còn lại.
+ Khoảng cách giữa hai thành phố có thể là khoảng cách địa lý, có thể là cước phí di chuyển hoặc thời gian di chuyển. Ta gọi chung là độ dài.

Hãy tìm một chu trình (một đường đi khép kín thỏa mãn các điều kiện trên) sao cho tổng độ dài chu trình là nhỏ nhất.

Bài toán này cũng được gọi là bài toán người du lịch. Một cách tổng quát, có thể không tồn tại một đường đi giữa hai thành phố a và b nào đó. Trong trường hợp đó ta cho một đường đi ảo giữa a và b với độ dài bằng ∞. Bài toán có thể biểu diễn bởi một đồ thị vô hướng có trọng số G=(V, E), trong đó mỗi thành phố được biểu diễn bởi một đỉnh, cạnh nối hai đỉnh biểu diễn cho đường đi giữa hai thành phố và trọng số của cạnh là khoảng cách giữa hai thành phố. Một chu trình đi qua tất cả các đỉnh của G, mỗi đỉnh một lần duy nhất, được gọi là chu trình Hamilton. Vấn đề là tìm một chu trình Hamilton mà tổng độ dài các cạnh là nhỏ nhất.

Dễ dàng thấy rằng, với phương pháp vét cạn ta xét tất cả các chu trình, mỗi chu trình tính tổng độ dài các cạnh của nó rồi chọn một chu trình có tổng độ dài nhỏ nhất. Tuy nhiên chúng ta phải xét tất cả $\frac{(n-1)!}{2}$ chu trình. Thực vậy, do mỗi chu trình đều đi qua tất cả các đỉnh (thành phố) nên ta có thể cố định một đỉnh. Từ đỉnh này ta có n-1 cạnh tới n-1 đỉnh khác, nên ta có n-1 cách chọn cạnh đầu tiên của chu trình. Sau khi đã chọn được cạnh đầu tiên, chúng ta còn n-2 cách chọn cạnh thứ hai, do đó ta có (n-1)(n-2) cách chọn hai cạnh. Cứ lý luận như vậy ta sẽ thấy có (n-1)! cách chọn một chu trình. Tuy nhiên với mỗi chu trình ta chỉ quan tâm đến tổng độ dài các cạnh chứ không quan tâm đến hướng đi theo chiều dương hay âm vì vậy có tất cả $\frac{(n-1)!}{2}$ phương án. Đó là một giải thuật có độ phức tạp là một thời gian mũ. Vì vậy khi áp dụng kỹ thuật tham ăn ở một số bài toán chúng ta chỉ có thể thu được các giải quyết tốt chứ không thể là tối ưu nhất.

Áp dụng kỹ thuật tham ăn vào bài toán này như sau :

Bước 1 : Sắp xếp các cạnh theo thứ tự tăng của độ dài.

Bước 2 : Xét các cạnh có độ dài từ nhỏ đến lớn để đưa vào chu trình.

Bước 3 : Một cạnh sẽ được đưa vào chu trình nếu cạnh đó thỏa hai điều kiện sau:

+ Không tạo thành một chu trình thiếu (không đi qua đủ n đỉnh)
+ Không tạo thành một đỉnh có cấp ≥ 3 (tức là không được có nhiều hơn hai cạnh xuất phát từ một đỉnh, do yêu cầu của bài toán là mỗi thành phố chỉ được đến một lần: một lần đến và một lần đi).

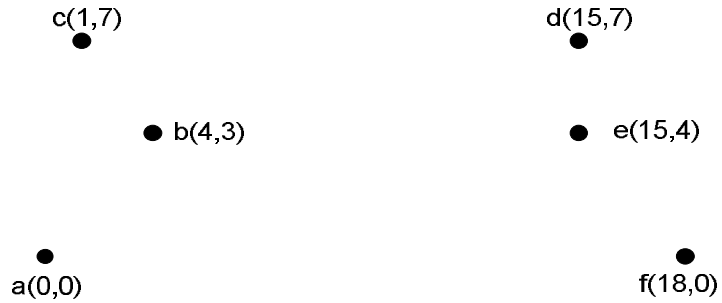
Bước 4 : Lặp lại bước 3 cho đến khi xây dựng được một chu trình.

Với kĩ thuật này ta chỉ cần n(n-1)/2 phép chọn nên ta có một giải thuật cần $O(n^2)$ thời gian.

Ví dụ 5.2 :

Cho bài toán TSP với 6 điểm có tọa độ tương ứng :

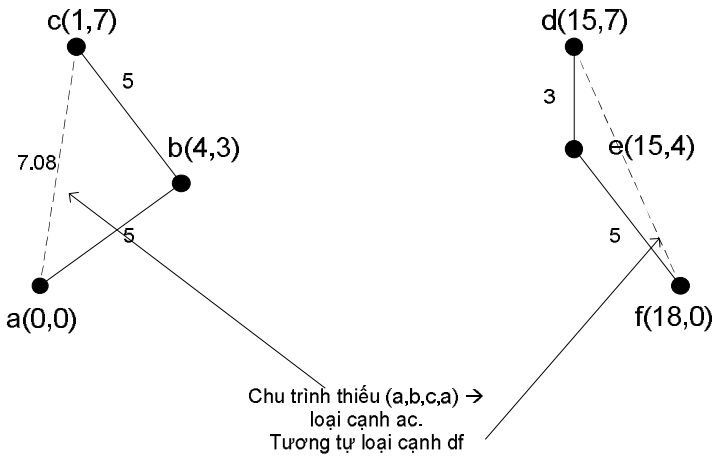
a(0,0), b(4,3), c(1,7), d(15,7), e(15,4) và f(18,0).



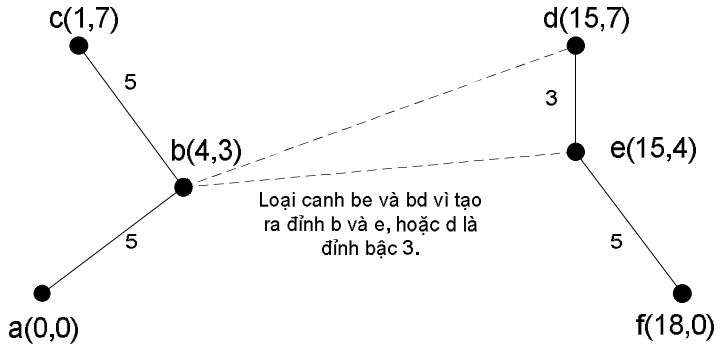
Do có 6 đỉnh nên có tất cả 15 cạnh. Đó là các cạnh: ab, ac, ad, ae, af, bc, bd, be, bf, cd, ce, cf, de, df và ef. Độ dài các cạnh ở đây là khoảng cách Euclide (khoảng cách 2 điểm A, B = $\sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}$).

stt	Cạnh	X1	Y1	X2	Y2	Độ dài
1	De	15	7	15	4	3
2	Ab	0	0	4	3	5
3	Bc	4	3	1	7	5
4	Ef	15	4	18	0	5
5	Ac	0	0	1	7	7.08
6	Df	15	7	18	0	7.62
7	Be	4	3	15	4	11.05
8	Bd	4	3	15	7	11.7
9	Cd	1	7	15	7	14
10	Bf	4	3	18	0	14.32
11	Ce	1	7	15	4	14.32

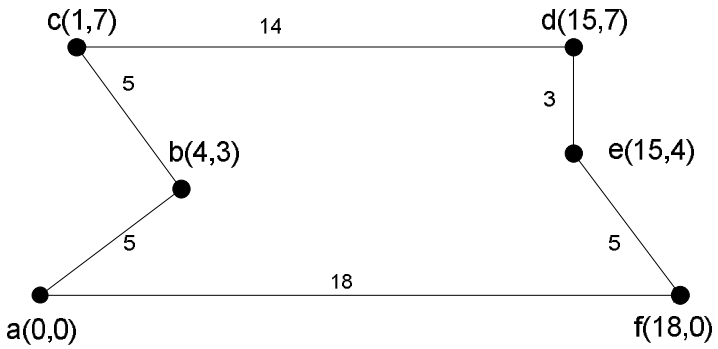
12	Ae	0	0	15	4	15.52
13	Ad	0	0	15	7	16.55
14	Af	0	0	18	0	18
15	Cf	1	7	18	0	18.38



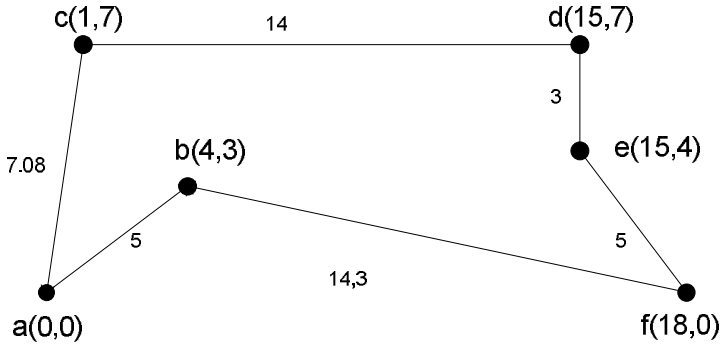
Trong số các cạnh thì canh de = 3 là nhỏ nhất, nên de được chọn vào chu trình. Kế đến là 3 cạnh ab, bc và ef đều có độ dài là 5. Cả 3 cạnh đều thỏa mãn hai điều kiện nói trên, nên đều được chọn vào chu trình. Cạnh có độ dài nhỏ kế tiếp là ac = 7.08, nhưng không thể đưa cạnh này vào chu trình vì nó sẽ tạo ra chu trình thiếu (a-b-c-a). Cạnh df cũng bị loại vì lý do tương tự. Cạnh be được xem xét nhưng rồi cũng bị loại do tạo ra đỉnh b và đỉnh e có cấp 3. Tương tự chúng ta cũng loại bd.



Cạnh cd là cạnh tiếp theo được xét và được chọn. Cuối cùng ta có chu trình a-b-c-d-e-f-a với tổng độ dài là 50. Đây chỉ là một phương án tốt nhưng chưa tối ưu.



Dựa vào giả thiết bài toán trên chúng ta có kết quả tối ưu của bài toán là chu trình a-c-d-e-f-b-a với tổng độ dài là 48.38.



Mã giả của giải thuật dành cho bài toán tìm đường đi của người giao hàng.

```
void TSP()
{
    /*E là tập hợp các cạnh, Chu_trinh là tập hợp các cạnh được chọn để đưa vào chu trình, mở đầu Chu_trinh rỗng*/
    /*Sắp xếp các cạnh trong E theo thứ tự tăng của độ dài*/
    Chu_Trinh = Φ;
    Gia = 0.0;
    while (E != Φ) {
        if (cạnh e có thể chọn)
        {
            Chu_Trinh = Chu_Trinh + [e];
            Gia = Gia + độ dài của e;
        }
        E= E-[e];
    }
}
```

Một cách tiếp cận khác của kĩ thuật tham ăn vào bài toán này là:

- Bước 1. Xuất phát từ một đỉnh bất kỳ, chọn một cạnh có độ dài nhỏ nhất trong tất cả các cạnh đi ra từ đỉnh đó để đến đỉnh kế tiếp.
- Bước 2. Từ đỉnh kế tiếp ta lại chọn một cạnh có độ dài nhỏ nhất đi ra từ đỉnh này thoả mãn hai điều kiện nói trên để đi đến dỉnh kế tiếp.
- Bước 3. Lặp lại bước 2 cho đến khi đi tới đỉnh n thì quay trở về đỉnh xuất phát.

5.2.3.2 Bài toán cái ba lô

Mô tả bài toán : Cho một cái ba lô có thể đựng một trọng lượng W và n loại đồ vật, mỗi đồ vật i có một trọng lượng g_i và một giá trị v_i. Tất cả các loại đồ vật đều có số lượng không hạn chế. Tìm một cách lựa chọn các đồ vật đựng vào ba lô, chọn các loại đồ vật nào, mỗi loại lấy bao nhiêu sao cho tổng trọng lượng không vượt quá W và tổng giá trị là lớn nhất.

Theo yêu cầu của bài toán thì ta cần những đồ vật có giá trị cao mà trọng lượng lại nhỏ để sao cho có thể mang được nhiều đồ có giá trị nhất, sẽ là hợp lý khi ta quan tâm đến yếu tố "đơn giá" của từng loại đồ vật tức là tỷ lệ giá trị/trọng lượng. Đơn giá càng cao thì đồ càng quý. Từ đó ta có kĩ thuật tham ăn để áp dụng cho bài toán này là:

- Bước 1 : Tính đơn giá cho các loại đồ vật.
- Bước 2 : Xét các loại đồ vật theo thứ tự đơn giá từ lớn đến nhỏ.
- Bước 3 : Với mỗi đồ vật được xét sẽ lấy một số lượng tối đa mà trọng lượng còn lại ba lô cho phép.
- Bước 4 : Xác định trọng lượng còn lại của ba lô và quay lại bước 3 cho đến khi không còn có thể chọn được đồ vật nào nữa.

Ví dụ 5.3:

Cho ba lô có trọng lượng là 37 và 4 loại đồ vật với trọng lượng và giá trị tương ứng được cho trong bảng sau đây:

Loại đồ vật	Trọng lượng	Giá trị
A	15	30
B	10	25
C	2	2
D	4	6

Từ bảng đã cho ta tính đơn giá cho các loại đồ vật và sắp xếp các loại đồ vật này theo thứ tự đơn giá giảm dần, ta có bảng sau:

Loại đồ vật	Trọng lượng	Giá trị	Đơn giá
B	10	25	2.5

A	15	30	2
D	4	6	1.5
C	2	2	1

Theo đó thì thứ tự ưu tiên để chọn đồ vật là B, A, D và cuối cùng là C.

Vật B được xét đầu tiên và ta chọn tối đa 3 cái vì mỗi cái vì trọng lượng mỗi cái là 10 và ba lô có trọng lượng 37. Sau khi đã chọn 3 vât loại B, trọng lượng còn lại trong ba lô là 37- 3*10 =7. Ta xét đến vật A, vì A có trọng lượng 15 mà trọng lượng còn lại của balô chỉ còn 7 nên không thể chọn vật A. Xét vật D và ta thấy có thể chọn 1 vật D, khi đó trọng lượng còn lại của ba lô là 7-4 = 3. Cuối cùng ta chọn được một vật C.

Như vậy chúng ta đã chọn 3 cái loại B, một cái loại D và 1 cái loại C. Tổng trọng lương là 3*10 + 1*4 + 1*2 = 36 và tổng giá trị là 3*25+1*6+1*2 = 83.

Giải thuật thô giải bài toán cái ba lô bằng kĩ thuật tham ăn như sau:

Tổ chức dữ liệu:

- Mỗi đồ vật được biểu diễn bởi một cấu trúc chứa các thông tin:

+ Ten: Lưu trữ tên đồ vật.

+ Trong_luong: Lưu trữ trọng lượng của đồ vật.

+ Gia_tri: Lưu trữ giá trị của đồ vật

+ Don_gia: Lưu trữ đơn giá của đồ vật

+ Phuong_an: Lưu trữ số lượng đồ vật được chọn theo phương án.

- Danh sách các đồ vật được biểu diễn bởi một mảng các đồ vật.

```
const int MAX_SIZE = 100;
struct Do_vat
{
     char Ten [20];
     float Trong_luong, Gia_tri, Don_gia;
     int Phuong_an;
};
 Do_vat dsdv[MAX_SIZE];
void Greedy (Do_vat  dsdv[], float W)
{
     /*Sắp xếp mảng dsdv theo thứ tự giảm của don_gia*/
     for (int i = 0; i < n; i++) {
          dsdv[i].Phuong_an  =  Chon(dsdv[i].Trong_luong,
W);
          W    =    W    –    dsdv[i].Phuong_an    *
dsdv[i].Trong_luong;
     }
}
```

Trong đó hàm Chon(trong_luong, W) nhận vào trọng lượng trong_luong của một vật và trọng lượng còn lại W của ba lô, trả về số lượng đồ vật được chọn, sao cho tổng trọng lượng của các vật được chọn không lớn hơn W. Nói riêng, trong trường

hợp trong_luong và W là hai sô nguyên thì Chon(Trong_luong, W) chính là W / Trong_luong.

Lưu ý:

Có một số biến thể của bài toán cái ba lô như sau:

1. Mỗi đồ vật i chỉ có một số lượng s_i. Với bài toán này khi lựa chọn vật i ta không được lấy một số lượng vượt quá s_i.

2. Mỗi đồ vật chỉ có một cái. Với bài toán này thì với mỗi đồ vật ta chỉ có thể chọn hoặc không chọn.

5.3 Kỹ thuật nhánh cận - Branch and Bound

5.3.1 Giới thiệu

Đối với các bài toán tìm phương án tối ưu, nếu chúng ta xét hết tất cả các phương án thì mất rất nhiều thời gian, nhưng nếu sử dụng kỹ thuật tham ăn thì phương án tìm được chưa hẳn đã là phương án tối ưu. Nhánh cận là kĩ thuật xây dựng cây tìm kiếm phương án tối ưu, nhưng không xây dựng toàn bộ cây mà sử dụng giá trị cận để hạn chế bớt các nhánh.

Cây tìm kiếm phương án có nút gốc biểu diễn cho tập tất cả các phương án có thể có, mỗi nút lá biểu diễn cho một phương án nào đó. Nút n có các nút con tương ứng với các khả năng có thể lựa chọn tập phương án xuất phát từ n. Kĩ thuật này gọi là phân nhánh.

Mỗi nút trên cây ta sẽ xác định một giá trị cận. Giá trị cận là một giá trị gần với giá của các phương án. Với bài toán tìm min ta sẽ xác định cận dưới còn với bài toán tìm max ta sẽ xác định cận trên. Cận dưới là giá trị nhỏ hơn hoặc bằng giá của phương án, ngược lại cận trên là giá trị lớn hơn hoặc bằng giá của phương án.

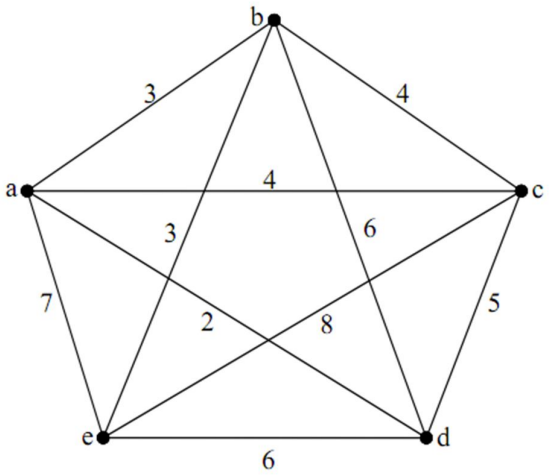
Chúng ta sẽ làm rõ kỹ thuật này thông qua bài toán tìm đường đi của người giao hàng.

5.3.2 Bài toán tìm đường đi của người giao hàng

Cây tìm kiếm phương án là cây nhị phân trong đó:

- Nút gốc là nút biểu diễn cho cấu hình bao gồm tất cả các phương án.
- Mỗi nút sẽ có hai con, con trái biểu diễn cho cấu hình bao gồm tất cả các phương án chứa một cạnh nào đó, con phải biểu diễn cho cấu hình bao gồm tất cả các phương án không chứa cạnh đó (các cạnh để xét phân nhánh được thành lập tuân theo một thứ tự nào đó, chẳng hạn thứ tự từ điển).
- Mỗi nút sẽ kế thừa các thuộc tính của tổ tiên của nó và có thêm một thuộc tính mới (chứa hay không chứa một cạnh nào đó).
- Nút lá biểu diễn cho một cấu hình chỉ bao gồm một phương án.
- Để quá trình phân nhánh mau chóng tới nút lá, tại mỗi nút ta cần có một quyết định bổ sung dựa trên nguyên tắc là mọi đỉnh trong chu trình đều có cấp 2 và không tạo ra một chu trình thiếu.

Ví dụ 5.3. Xét bài toán tìm đường đi của người giao hàng có 5 đỉnh với độ dài các cạnh được cho như sau :



- Các cạnh theo thứ tự từ điển để xét là: ab, ac, ad, ae, bc, bd, be, cd, ce và de.
- Giả sử người giao hàng xuất phát từ đỉnh A, khi đó cây phương án có nút gốc là A và bao gồm tất cả các phương án.
- Hai con của A là B và C, trong đó B bao gồm tất cả các phương án chứa cạnh ab, C bao gồm tất cả các phương án không chứa ab, kí hiệu là $\overline{ab}$.
- Hai con của B là D và E. Nút D bao gồm tất cả các phương án chứa ac. Vì các phương án này vừa chứa ab (kế thừa của B) vừa ab

5.4 Kỹ thuật quy hoạch động - Dynamic programming

5.4.1 Giới thiệu

Khi tìm hiểu kỹ thuật chia để trị, bài toán thường bị dẫn đến một phương án có giả thuật đệ quy. Trong số các giải thuật đó, có thể có một số giải thuật có độ phức tạp thời gian mũ. Tuy nhiên, thường chỉ có một số đa thức các bài toán con, điều đó có nghĩa là chúng ta đã phải giải một số bài toán con nào đó nhiều lần. Để tránh việc giải dư thừa một số bài toán con, chúng ta tạo ra một bảng để lưu trữ kết quả của các bài toán con và khi cần chúng ta sẽ sử dụng kết quả đã được lưu trong bảng mà không cần phải giải lại bài toán đó. Lấp đầy bảng kết quả các bài toán con theo một quy luật nào đó để nhận được kết quả của bài toán ban đầu (cũng đã được lưu trong một số ô nào đó của bảng) được gọi là quy hoạch động.

Tóm tắt giải thuật quy hoạch động :

- Bước 1 : Tạo bảng bằng cách:

+ Bước 1.1 : Gán giá trị cho một số ô nào đó.

+ Bước 1.2 : Gán trị cho các ô khác nhờ vào giá trị của các ô trước đó.

- Bước 2 : Tra bảng và xác định kết quả của bài toán ban đầu.

Ưu điểm của kỹ thuật quy hoạch động là chương trình thực hiện nhanh do không phải tốn thời gian giải lại một bài toán con đã được giải. Kĩ thuật quy hoạch động có thể vận dụng để giải các bài toán tối ưu, các bài toán có công thức truy hồi.

Phương pháp quy hoạch động sẽ không đem lại hiệu quả trong các trường hợp sau:

- Không tìm được công thức truy hồi.
- Số lượng các bài toán con cần giải quyết và lưu giữ kết quả là rất lớn.

- Sự kết hợp lời giải của các bài toán con chưa chắc cho ta lời giải của bài toán ban đầu.

Sau đây là một số bài toán điển hình có thể giải bằng kỹ thuật quy hoạch động

5.4.2 Một số bài toán minh họa

5.4.2.1 Bài toán tính số tổ hợp

Một bài toán khá quen thuộc là tính số tổ hợp chập k của n theo công thức truy hồi:

$$C_n^k = \begin{cases} 1 \, nếu \, k = 0 \, hoặc \, k = n \\ C_{n-1}^{k-1} + C_{n-1}^{k} \, nếu \, 0 < k < n \end{cases}$$

Dựa vào công thức truy hồi trên, chúng ta có giải thuật đệ quy cho bài toán như sau :

```
int TinhToHop (int n, int k)
{
     if(k==0 || k==n)
          return 1;
     else
          return TinhToHop (n-1,k-1) + TinhToHop (n-1,k);
}
```

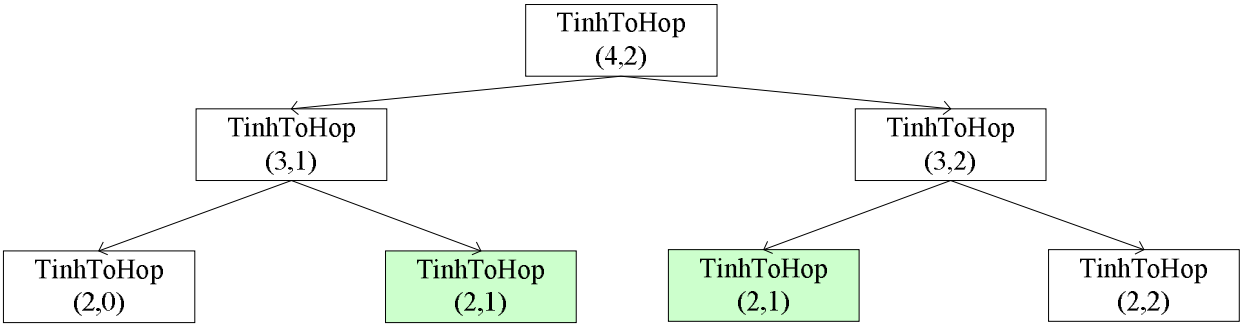
Gọi T(n) là thời gian để tính số tổ hợp chập k của n, thì ta có phương trình đệ quy:

T(1) = C1 và T(n) = 2T(n-1) + C2.

n

Giải phương trình này ta được T(n) = O(2^n), như vậy là một giải thuật thời gian mũ, trong khi chỉ có một đa thức các bài toán con. Điều đó chứng tỏ rằng có những bài toán con được giải nhiều lần.

Chẳng hạn để tính TinhToHop (4,2) ta phải tính TinhToHop (3,1) và TinhToHop (3,2). Để tính TinhToHop (3,1) ta phải tính TinhToHop (2,0) và TinhToHop (2,1). Để tính TinhToHop (3,2) ta phải tính TinhToHop (2,1) và TinhToHop (2,2). Như vậy để tính TinhToHop (4,2) ta phải tính TinhToHop (2,1) hai lần.



Hình 2. Sơ đồ thực hiện tinhToHop(4,2)

Áp dụng kĩ thuật quy hoạch động để khắc phục tình trạng trên, ta xây dựng một bảng gồm n+1 dòng (từ 0 đến n) và n+1 cột (từ 0 đến n) và điền giá trị cho O(i,j) theo quy tắc sau: (Quy tắc tam giác Pascal):

- C(0,0) = 1;
- C(i,0) =1;
- C(i,i) = 1 với $0 < i \leq n$;
- C(i,j) =C(i-1,j-1) + C(i-1,j) với $0 < j < i \leq n$.

Với O(n,k) là tinhToHop (n,k)

Cụ thể với n=4 ta có bảng như sau :

i \ j	0	1	2	3	4
0	1				
1	1	1			
2	1	2	1		
3	1	3	3	1	
4	1	4	6	4	1

Giải thuật quy hoạch động của bài toán tính tổ hợp :

```
int tinhToHop_DP (int n, int k)
{
        int C[n+1][n+1];
/*1*/   C[0][0] = 1;
/*2*/   for (int i = 1; i <= n; i++)
        {
/*3*/     C[i][0] = 1;
/*4*/     C[i][i] = 1;
/*5*/     for (int j = 1; j < i; j++)
              C[i][j]=C[i-1][j-1]+C[i-1][j];
        }
/*6*/   return C[n][k];
}
```

- Vòng lặp /*5*/ thực hiện i-1 lần, mỗi lần O(1).

- Vòng lặp /*2*/ có i chạy từ 1 đến n, nên nếu gọi T(n) là thời gian thực hiện giải thuật thì ta có: $T(n) = \sum_{i=1}^{n}(i-1) = \frac{n(n-1)}{2} = O(n^2)$.

Như vậy, qua việc xác định độ phức tạp, ta thấy rõ giải thuật quy hoạch động hiệu quả hơn nhiều so với giải thuật đệ quy ($n^2 < 2^n$). Tuy nhiên việc dùng bảng (mảng 2 chiều) như trên còn lãng phí ô nhớ, vì có gần 1 nửa số ô nhớ không cần dùng đến. Do đó chúng ta sẽ cải tiến thêm một bước bằng cách sử dụng véc-tơ (mảng 1 chiều) để lưu kết quả trung gian. Cách cải tiến như sau :

Dùng một véc-tơ V có n+1 phần tử từ V[0] đến V[n]. Véc-tơ V sẽ lưu trữ các giá trị tương ứng với dòng i trong tam giác Pascal ở trên. Trong đó V[j] lưu trữ giá trị số tổ hợp chập j của i (C_i^j) (j = 0 đến i). Dĩ nhiên do chỉ có một véctơ V mà phải lưu trữ nhiều dòng i do đó tại mỗi bước, V chỉ lưu trữ được một dòng và ở bước cuối cùng, V lưu trữ các giá trị ứng với i = n, trong đó V[k] chính là C_n^k.

Ban đầu, ứng với i =1, ta cho V[0] = 1 và V[1] = 1. Tức là $C_1^0 = 1$ *và* $C_1^1 = 1$. Với các giá trị i từ 2 đến n, ta thực hiện như sau :

V[0] = 1, tức là $C_i^0 = 1$. Tuy nhiên giá trị V[0] = 1 đã được gán ở trên nên không cần gán lại.

Với j từ 1 đến i-1, ta vẫn áp dụng công thức $C_i^j = C_{i-1}^{j-1} + C_{i-1}^j$, nghĩa là để tính các giá trị dòng i, ta phải dựa vào dòng i-1. Nhưng do chỉ có 1 véc-tơ V và lúc này nó lưu giá trị dòng i, còn dòng i-1 không còn. Để khắc phục điều này ta dùng thêm 2 biến trung gian p1 và p2. Trong đó p1 = C_{i-1}^{j-1} và p2 = C_{i-1}^j. Khởi đầu p1 = V[0] (=C_{i-1}^0) và p2=V[j] (=C_{i-1}^j). Sau đó V[j] lưu giá trị C_i^j sẽ được gán bởi p1+p2. Tiếp theo p1 được gán bởi p2, nghĩa là khi j tăng lên 1 đơn vị thành j+1 thì p1 là C_{i-1}^j và nó được dùng để tính C_i^{j+1}.

Cuối cùng với j=i ta gán V[i] = 1 (=C_i^i=1).

Giải thuật cải tiến như sau :

```
int tinhToHop_DP_CaiTien (int n, int k)
{
     int V[n+1];
     int p1, p2;
/*1*/     V[0] = 1;
/*2*/     V[1] = 1;
/*3*/     for (int i = 2; i <= n; i++)
     {
/*4*/          p1 = V[0];
/*5*/          for (int j = 1; j < i; j++)
          {
/*6*/               p2 = V[j];
/*7*/               V[j]= p1+p2;
/*8*/               p1= p2;
```

```
        }
/*9*/           V[i] = 1;
}
/*10*/    return V[k];
}
```

Độ phức tạp của giải thuật vẫn là $O(n^2)$.

5.4.3 Bài toán ba lô.

Xét lại bài toán cái ba lô phần 5.2.3.2 như sau:

Cho một cái ba lô có thể đựng một trọng lượng W và n loại đồ vật, mỗi đồ vật i có một trọng lượng g_i và một giá trị v_i. Tất cả các loại đồ vật đều có số lượng không hạn chế. Tìm một cách lựa chọn các đồ vật đựng vào ba lô, chọn các loại đồ vật nào, mỗi loại lấy bao nhiêu sao cho tổng trọng lượng không vượt quá W và tổng giá trị là lớn nhất.

Ta sử dụng kỹ thuật quy hoạch động để giải với các số liệu được cho đều là số nguyên.

Giả sử:

- X[k,V]: là số đồ vật k được chọn.
- F[k,V]: là tổng giá trị của k đồ vật đã được chọn.
- V là trọng lượng còn lại của ba lô, k=1..n, V=1..W.

Trong trường hợp đơn giản nhất, khi chỉ có một đồ vật, ta tính được X[1,V] và F[1,V] với mọi V từ 1 đến W là: $X[1,V] = V/g_1$ và $F[1,V] = X[1,V] * v_1$.

Giả sử ta đã tính được F[k-1,V], khi có thêm đồ vật thứ k, ta sẽ tính được F[k,V], với mọi V từ 0 đến W. Cách tính như sau:

- Nếu ta chọn x_k đồ vật loại k, thì trọng lượng còn lại của ba lô dành cho k-1 đồ vật từ 1 đến k-1 là $U = V-x_k*g_k$.
- Tổng giá trị của k loại đồ vật đã được chọn $F[k,V] = F[k-1,U] + x_k*v_k$, với x_k thay đổi từ 0 đến y_k ($y_k = V/g_k$) và ta sẽ chọn x_k sao cho F[k,V] lớn nhất.

Ta có công thức truy hồi như sau:

- $X[1,V] = V / g_1$ và $F[1,V] = X[1,V] * v_1$.
- $F[k,V] = Max (F[k-1,V-x_k*g_k] + x_k*v_k)$ với x_k chạy từ 0 đến V/g_k.
- Sau khi xác định được F[k,V] thì X[k,V] là x_k ứng với giá trị F[k,V] được chọn trong công thức trên.

Để lưu các giá trị trung gian trong quá trình tính F[k,V] theo công thức truy hồi trên, ta sử dụng một bảng gồm n dòng từ 1 đến n, dòng thứ k ứng với đồ vật loại k và W+1 cột từ 0 đến W, cột thứ V ứng với trọng lượng V. Mỗi cột V bao gồm hai cột nhỏ, cột bên trái lưu F[k,V], cột bên phải lưu X[k,V]. Trong lập trình ta sẽ tổ chức hai bảng tách rời là F và X.

Ví dụ 5.5: Xét bài toán cái ba lô với trọng lượng W=9, và 5 loại đồ vật được cho trong bảng sau:

Đồ vật	Trọng lượng (g_i)	Giá trị (v_i)
1	3	4
2	4	5
3	5	6
4	2	3
5	1	1

Ta có bảng F[k,V] và X[k,V] như sau, trong đó mỗi cột V có hai cột con, cột bên trái ghi F[k,V] và cột bên phải ghi X[k,V].

v / k	0		1		2		3		4		5		6		7		8		9	
1	0	0	0	0	0	0	4	1	4	1	4	1	8	2	8	2	8	2	12	3
2	0	0	0	0	0	0	4	0	5	1	5	1	8	0	9	1	10	2	12	0
3	0	0	0	0	0	0	4	0	5	0	6	1	8	0	9	0	10	0	12	0
4	0	0	0	0	3	1	4	0	6	2	7	1	9	3	10	2	12	4	13	3
5	0	0	1	1	3	0	4	0	6	0	7	0	9	0	10	0	12	0	13	0

Trong bảng trên, việc điền giá trị cho dòng 1 rất đơn giản bằng cách sử dụng công thức: $X[1,V] = V/g_1$ và $F[1,V]=X[1,V]*v_1$.

Từ dòng 2 đến dòng 5, phải sử dụng công thức truy hồi:

$F[k,V] = Max (F[k-1,V-x_k*g_k] + x_k *v_k)$ với x_k chạy từ 0 đến V/g_k.

Chẳng hạn để tính F[2,7] ta có x_k chạy từ 0→ V/g_k, trong trường hợp này là x_k chạy từ 0→ 7/4. Vậy x_k có hay giá trị 0 và 1.

Khi đó F[2,7] = Max (F[2-1, 7-0*4] + 0*5, F[2-1,7-1 *4] + 1*5).

= Max(F[1,7], F[1,3] + 5) = Max(8, 4+5) = 9.

F[2,7] = 9 ứng với $x_k = 1$, do đó X[2,7] =1.

Vấn đề bây giờ là cần phải tra trong bảng trên để xác định phương án.

Ban đầu trong lượng còn lại của ba lô V=W.

Xét các đồ vật từ n đến 1, với mỗi đồ vật k, ứng với trọng lượng còn lại V của ba lô nếu X[k,V] > 0 thì chọn X[k,V] đồ vật loại k. Tính lại $V = V - X[k,V] * g_k$.

Ví dụ, trong bảng trên, ta sẽ xét các đồ vật từ 5 đến 1. Khởi đầu V = W = 9.

Với k = 5, vì X[5,9] = 0 nên ta không chọn đồ vật loại 5.

Với k = 4, vì X[4,9] = 3 nên ta chọn 3 đồ vật loại 4. Tính lại V = 9-3*2 = 3.

Với k = 3, vì X[3,3] = 0 nên ta không chọn đồ vật loại 3.

Với k = 2, vì X[2,3] = 0 nên ta không chọn đồ vật loại 2.

Với k = 1, vì X[1,3] = 1 nên ta chọn 1 đồ vật loại 1. Tính lại V = 3 – 1 * 3 = 0.

Vậy tổng trọng lương các vật được chọn là 3 * 2 + 1 * 3 = 9. Tổng giá trị các vật được chọn là 3 * 3 + 1 * 4 = 13.

Giải thuật theo kĩ thuật quy hoạch động như sau:

Phần tổ chức dữ liệu:

- Mỗi đồ vật được biểu diễn bởi một cấu trúc (struct) gồm các thông tin:
  + Ten: Lưu trữ tên đồ vật.
  + Trong_luong: Lưu trữ trọng lượng của đồ vật.
  + Gia_tri: Lưu trữ giá trị của đồ vật
  + Phuong_an: Lưu trữ số lượng đồ vật được chọn theo phương án.
- Danh sách các đồ vật được biểu diễn bởi một mảng các đồ vật.
- Bảng được biểu diễn bởi một mảng hai chiều các số nguyên để lưu trữ các giá trị F[k,v] và X[k,v].

Phần cài đặt bằng C/C++:

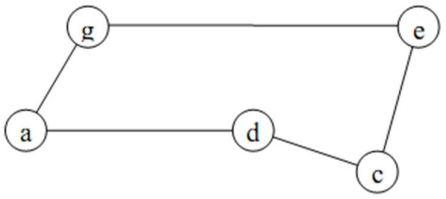
```
const int MAX =10;
struct Do_vat{
   char Ten[20];
   int Trong_luong,
    int Gia_tri;
   int Phuong_an;
};
typedef Do_vat Danh_sach_vat [MAX]; // định nghĩa mảng 1 chiều kiểu Do_vat bằng tên
                                    //Danh_sach_vat
typedef int Bang[10][100]           // định nghĩa mảng 2 chiều kiểu int bằng tên
                                    //Bang
void Tao_Bang (Danh_sach_vat ds_vat, int n, int W, Bang F, Bang X)
 {
  int xk, yk, k;
  int FMax, XMax, v;
  /*Lấp đầy hàng đầu tiên của 2 bảng*/
  for (v= 0; v <= W; v++)
   {
       X[1][v] = v/ds_vat[1].trong_luong;
           F[1][v] = X[1] [v] * ds_vat[1].gia_tri;
  }
  /*Lấp đầy các hàng còn lại*/
  for (k = 2; k <= n; k++)
   {
        X[k][0] = 0;
        F[1][0] = 0;
   for ( v=1; v <= W; v++)
   {
        Fmax = F[k-1][v] ;
        XMax = 0;
        yk = v/ds_vat[k].trong_luong;
        for (xk=1; xk <= yk; xk++)
            if ( F[k-1][v-xk * ds_vat[k].trong_luong] + xk
* ds_vat[k].gia_tri >FMax)
        {
```

```
                    FMax=F[k-1][v-k*ds_vat[k].trong_luong]  +
xk*ds_vat[k].gia_tri;
                    XMax= xk;
          }
          F[k][v] = FMax;
          X[k][v] = XMax;
             }
    }
}
//-------------------------------------------------------
// hàm tra bảng
void Tra_Bang(Danh_sach_vat ds_vat, int n, int W Bang F,
Bang X)
{
  int k, v;
  v = W;
  for (k = n; k >=1; k--)
       if (X[k][v] > 0)
        {
            ds_vat[k].Phuong_an := X[k][v];
            v = v - X[k][ v] * ds_vat[k].trong_luong;
       }
}
```

Bài toán đường đi của người giao hàng

Xét lại bài toán tìm đường đi của người giao hàng ở mục 5.2.3.1, chúng ta áp dụng kỹ thuật quy hoạch động để giải như sau.

Đặt S = {x_1, x_2, …, x_k} là tập hợp con các cạnh của đồ thị G = (V, E). Ta nói rằng một đường đi P từ v đến w phủ lên S nếu P = {v, x_1, x_2, …, x_k, w}, trong đó x_i có thể xuất hiện ở một thứ tự bất kì, nhưng chỉ xuất hiện duy nhất một lần. Ví dụ đường cho trong hình sau, đi từ a đến a, phủ lên {c, d, e, g}.



Ta định nghĩa d(v, w, S) là tổng độ dài của đường đi ngắn nhất từ v đến w, phủ lên S. Nếu không có một đường đi như vậy thì đặt d(v, w, S) = ∞. Một chu trình Hamilton nhỏ nhất C_{min} của G phải có tổng độ dài là c(Cmin) = d(v,v, V - {v}). Trong đó v là một đỉnh nào đó của V. Ta xác định C_{min} như sau:

Nếu |V| = 1 (G chỉ có một đỉnh) thì c(C_{min}) = 0, ngược lại ta có công thức đệ qui để tính d(v, w, S) là:

d(v,w,{}) = c(v,w)

d(v,w,S) = min {c(v, x) + d(x, w, S – {x}}, lấy $\forall x \in S$

Trong đó c(v,w) là độ dài của cạnh nối 2 đỉnh v và w nếu nó tồn tại hoặc là ∞ nếu ngược lại. Dòng thứ 2 trong công thức đệ qui ứng với tập $S \neq \emptyset$, nó chỉ ra rằng

đường đi ngắn nhất từ v đến w phủ lên S, trước hết phải đi đến một đỉnh x nào đó trong S và sau đó là đường đi ngắn nhất từ x đến w, phủ lên tập S-{x}.

Bằng cách lưu trữ các đỉnh x trong công thức đệ qui nói trên, chúng ta sẽ thu được một chu trình Hamilton tối tiểu.

BÀI TẬP CHƯƠNG 5

Bài 1. Giả sử có hai đội A và B tham gia một trận thi đấu thể thao, đội nào thắng trước n hiệp thì sẽ thắng cuộc. Chẳng hạn một trận thi đấu bóng chuyền 5 hiệp, đội nào thắng trước 3 hiệp thì sẽ tháng cuộc. Giả sử hai đội ngang tài ngang sức. Đội A cần thắng thêm i hiệp để thắng cuộc còn đội B thì cần thắng thêm j hiệp nữa. Gọi P(i,j) là xác suất để đội A cần i hiệp nữa để chiến thắng, B cần j hiệp. Dĩ nhiên i,j đều là các số nguyên không âm.

Để tính P(i,j) ta thấy rằng nếu i=0, tức là đội A đã thắng nên P(0,j) = 1. Tương tự nếu j=0, tức là đội B đã thắng nên P(i,0) = 0. Nếu i và j đều lớn hơn không thì ít nhất còn một hiệp nữa phải đấu và hai đội có khả năng "5 ăn, 5 thua" trong hiệp này.

Như vậy P(i,j) là trung bình cộng của P(i-1,j) và P(i,j-1). Trong đó P(i-1,j) là xác suất để đội A thắng cuộc nếu nó thắng hiệp đó và P(i,j-1) là xác suất để A thắng cuộc nếu nó thua hiệp đó. Tóm lại ta có công thức tính P(i,j) như sau:

P(i,j) = 1 Nếu i = 0

P(i,j) = 0 Nếu j = 0

P(i,j) = (P(i-1,j) + P(i,j-1))/2 Nếu i > 0 và j > 0.

a) Viết một hàm đệ quy để tính P(i,j). Tính độ phức tạp của hàm đó.
b) Dùng kĩ thuật quy hoạch động để viết hàm tính P(i,j). Tính độ phức tạp của hàm đó.
c) Viết hàm P(i,j) bằng kĩ thuật quy hoach động nhưng chỉ dùng mảng 1chiều (để tiết kiệm bộ nhớ).

Bài 2:

Bài toán phân công lao động: Có n công nhân có thể làm n công việc. Công nhân i làm công việc j trong một khoảng thời gian t_{ij}. Phải tìm một phương án phân công như thế nào để các công việc đều được hoàn thành, các công nhân đều có việc làm, mỗi công nhân chỉ làm một công việc và mỗi công việc chỉ do một công nhân thực hiện đồng thời tổng thời gian là nhỏ nhất.

1. Mô tả kĩ thuật tham ăn cho bài toán phân công lao động.

2. Tìm phương án theo giải thuật tham ăn cho bài toán phân công lao động được cho trong bảng sau. Trong đó mỗi dòng là một công nhân, mỗi cột là một công việc ô (i,j) ghi thời gian t_{ij} mà công nhân i cần để hoàn thành công việc j. (Cần chỉ rõ công nhân nào làm công việc gì và tổng thời gian là bao nhiêu).

Công việc / Công nhân	1	2	3	4	5
1	5	6	4	7	2
2	5	2	4	5	1
3	4	5	4	6	3
4	5	5	3	4	2
5	3	3	5	2	5

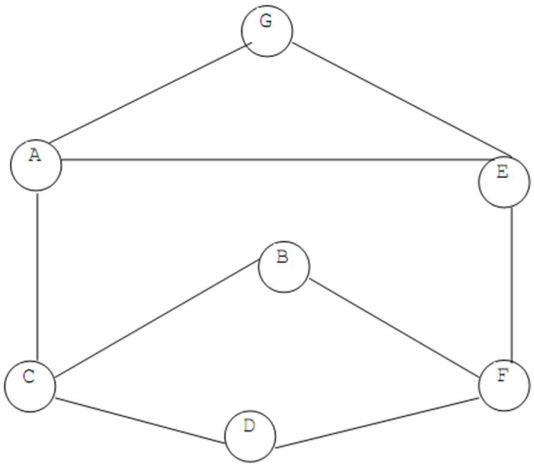
Bài 3.

Người ta muốn tô màu bản đồ các nước trên thế giới, mỗi nước đều được tô màu và hai nước có biên giới chung nhau thì không được có màu giống nhau (các nước không chung biên giới có thể được tô màu giống nhau). Tìm một phương án tô màu sao cho số loại màu phải dùng ít nhất.

Người ta có thể mô hình hóa bản đồ thế giới bằng một đồ thị không có hướng, trong đó mỗi đỉnh biểu diễn cho một nước, biên giới của hai nước được biểu diễn bằng cạnh nối hai đỉnh. Bài toán tô màu bản đồ thế giới trở thành bài toán tô màu các đỉnh của đồ thi: Mỗi đỉnh của đồ thị phải được tô màu và hai đỉnh có chung một cạnh thì không được tô cùng một màu (các đỉnh không chung cạnh có thể được tô cùng một màu). Tìm một phương án tô màu sao cho số loại màu phải dùng là ít nhất.

1. Hãy mô tả kĩ thuật tham ăn để giải bài toán tô màu cho đồ thị.

2. Áp dụng kĩ thuật tham ăn để tô màu cho các đỉnh của đồ thị sau (các màu có thể sử dụng để tô là: ĐỎ, CAM, VÀNG, XANH, ĐEN, NÂU, TÍM)



Bài 4. Cho bài toán cái ba lô với trọng lượng của ba lô W = 30 và 5 loại đồ vật được cho trong bảng bên dưới. Tất cả các loại đồ vật đều chỉ có một cái.

Loại đồ vật	Trọng lượng	Giá trị
A	15	30
B	10	25
C	2	2
D	4	6
E	8	24

1. Giải bài toán bằng kỹ thuật tham ăn.

2. Viết chương trình tìm phương án cho bài toán.

Bài 5. Cho n loại hàng. Món hàng thuộc loại hàng i cókhối lượng A[i] và giá trị C[i]. Số lượng các món hàng của mỗi loại không hạn chế. Cần chọn các món hàng trong từng loại để bỏ vào một ba lô sao cho tổng giá trị của các món hàng đã chọn

là lớn nhất nhưng tổng khối lượng của chúng không vượt quá khối lượng M cho trước. Cho biết số lượng món hàng từng loại hàng được chọn.

Ví dụ: n = 5, M = 13

i	1	2	3	4	5
A[i]	3	4	5	2	1
C[i]	4	5	6	3	1

Tổng giá trị của các món hang bỏ vào ba lô: 19.

Các món được chọn:

+ 1 gói hàng loại 1 có khối lượng 3 và giá trị 4.

+ 5 gói hàng loại 4 có khối lượng 2 và giá trị 3.

1. Xác định công thức đệ quy:

Gọi F(i,v) là tổng giá trị lớn nhất của các món hàng được chọn sao cho tổng khối lượng <= v trong i loại hàng.

- Trường hợp A[i] > v: F(i, v) = F(i-1, v).
- Trường hợp A[i] <= v:

+ Nếu loại hàng i không được chọn thì: F(i, v) = F(i-1, v)

+ Nếu có k món hàng loại i được chọn: (1 <= k <= v/A[i])

F(i, v) = F(i-1, v – A[i]*k) + C[i]*k

Do đó: F(i, v) = Max{F(i-1, v –A[i]*k) + C[i]*k } k ϵ [0, v/A[i]].

- Bài toán nhỏ nhất ứng với i = 0 hay v=0 ta có: F(0, v) = 0.

2. Công thức đệ quy:

Gọi F(i, v) là tổng giá trị lớn nhất của các món hàng được chọn sao cho tổng khối lượng<= v trong i loại hàng.

- Trường hợp A[i] > v: F(i, v) = F(i-1, v).
- Trường hợp A[i] <= v:

+ Nếu loại hàng i không được chọn thì: F(i, v) = F(i-1, v)

+ Nếu có k món hàng loại i được chọn: (1 <= k <= v/A[i])

F(i, v) = F(i-1, v – A[i]*k) + C[i]*k

Do đó:

F(i, v) = Max{F(i-1, v –A[i]*k) + C[i]*k } k ∈[0, v/A[i]]

- Bài toán nhỏ nhất ứng với i = 0 hay v=0 ta có: F(0, v) = 0

Gọi F(i, v) là tổng giá trị lớn nhất của các món hàng được chọn có tổng khối lượng <= v trong i loại hàng đầu tiên.

- Với i = 0 : F(i, v) = 0
- Với i > 0 :

+ A[i] > v : F(i, v) = F(i -1, v)

+ A[i] <= v : F(i, v) = Max{ F(i-1, v –A[i]*k) + C[i]*k }
với k ∈[0, v/A[i]]

Xây dựng bảng phương án

- Cấu trúc bảng phương án: dùng 2 mảng

  + Mảng F[0..n][0..M]: F[i][v] chứa giá trị của các F(i, v).

  + Mảng S[1..n][1..M]: S[i][v] chứa số món hàng loại i được chọn.

- Nếu F(i, v) = F(i - 1, v): S[i][v] = 0.
- Ngược lại S[i][v] = k
- Cách tính giá trị trên bảng phương án:

  + Điền số 0 cho các ô trên dòng 0 và cột 0 của bảng F.

  + Sử dụng công thức đệ quy và giá trị trên dòng i -1 để tính dòng i của bảng F và bảng S.

Ví dụ :

Lập bảng phương án F: F[i][v]

Với i > 0 :

- A[i] > v : F(i, v) = F(i -1, v)
- A[i] <= v : F(i, v) = Max{ F(i-1, v –A[i]*k) + C[i]*k } với k ∈[0, v/A[i]]

C[i]	A[i]	i \ v	0	1	2	3	4	5	6	7	8	9	10	11	12	13
		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	3	1	0	0	0	4	4	4	8	8	8	12	12	12	16	16
5	4	2	0	0	0	4	4	5	8	9	9	12	13	14	16	17
6	5	3	0	0	0	4	4	5	8	9	10	12	13	14	16	17
3	2	4	0	0	0	4	4	7	8	10	11	13	14	16	17	19
1	1	5	0	0	1	4	5	7	8	10	11	13	14	16	17	19

Lập bảng phương án S: (S[i][v])

C[i]	A[i]	i \ v	1	2	3	4	5	6	7	8	9	10	11	12	13
4	3	1	0	0	1	1	1	2	2	2	3	3	3	4	4
5	4	2	0	0	0	0	1	0	1	1	0	1	2	0	1
6	5	3	0	0	0	0	0	0	0	1	0	0	0	0	0
3	2	4	0	0	0	0	1	0	2	1	3	2	4	3	5
1	1	5	0	1	0	1	0	0	0	0	0	0	0	0	0

Truy vết tìm lại các gói hàng đã chọn

Bắt đầu từ ô S[n][M] trên dòng n ta dò ngược về dòng 1 theo nguyên tắc:

- Nếu S[i][v] ≠0 thì:

  + Loại hàng i được chọn với số lượng là S[i][v]

+ Truy tiếp ô S[i-1][v -S[i][v]*A[i]].

- Nếu S[i][v] = 0 thì:

+ Loại hàng i không được chọn,

+ Truy tiếp ô S[i-1][v].

C[i]	A[i]	i \ v	1	2	3	4	5	6	7	8	9	10	11	12	13
4	3	1	0	0	1	1	1	2	2	2	3	3	3	4	4
5	4	2	0	0	0	0	1	0	1	1	0	1	2	0	1
6	5	3	0	0	0	0	0	0	0	1	0	0	0	0	0
3	2	4	0	0	0	0	1	0	2	1	3	2	4	3	5
1	1	5	0	1	0	1	0	0	0	0	0	0	0	0	0

Yêu cầu: Hãy xây dựng thuật toán và viết chương trình minh họa:

a) Thuật toán tạo bảng phương án.
b) Thuật toán truy vết tìm lại các gói hàng đã chọn.

Bài 6. Bài toán xếp lịch thi đấu thể thao

Kỹ thuật chia để trị không những chỉ có ứng dụng trong thiết kế giải thuật mà còn trong nhiều lĩnh vực khác của cuộc sống. Chẳng hạn xét việc xếp lịch thi đấu thể thao theo thể thức đấu vòng tròn 1 lượt cho n đấu thủ. Mỗi đấu thủ phải đấu với các đấu thủ khác, và mỗi đấu thủ chỉ đấu nhiều nhất một trận mỗi ngày. Yêu cầu là xếp một lịch thi đấu sao cho số ngày thi đấu là ít nhất.

Ta dễ dàng thấy rằng tổng số trận đấu của toàn giải là $\frac{n(n-1)}{2}$. Như vậy nếu n là một số chẵn thì ta có thể sắp n/2 cặp thi đấu trong một ngày và do đó cần ít nhất n-1 ngày. Ngược lại nếu n là một số lẻ thì n-1 là một số chẵn nên ta có thể sắp (n-1)/2 cặp thi đấu trong một ngày và do đó cần n ngày.

Giả sử $n = 2^k$ thì n là một số chẵn và do đó cần tối thiểu n-1 ngày. Lịch thi đấu là một bảng n dòng và n-1 cột. Các dòng được đánh số từ 1 đến n và các cột được đánh số từ 1 đến n-1, trong đó dòng i biểu diễn cho đấu thủ i, cột j biểu diễn cho ngày thi đấu j và ô(i,j) ghi đấu thủ phải thi đấu với đấu thủ i trong ngày j.

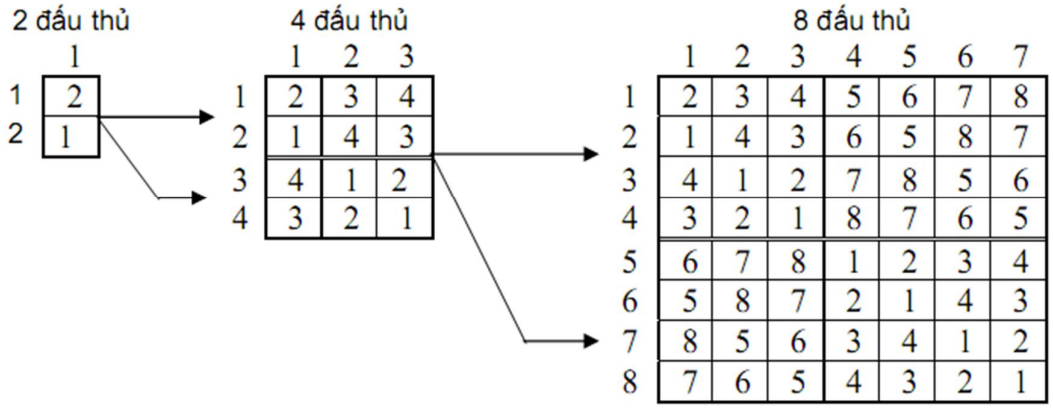
Kỹ thuật chia để trị xây dựng lịch thi đấu như sau: để sắp lịch cho n đấu thủ, ta sẽ sắp lịch cho n/2 đấu thủ, để sắp lịch cho n/2 đối thủ, ta sắp lịch cho n/4 đối thủ,…Quá trình đệ quy đến bài toán cơ sở là sắp lịch thi đấu cho 2 đối thủ. Hai đấu thủ này sẽ thi đấu 1 trận trong 1 ngày, lịch thi đấu cho họ thật đơn giản. Tuy nhiên, việc khó khăn chính là ở chỗ từ các lịch thi đấu cho hai đấu thủ, ta tổng hợp lại để được lịch thi đấu của 4 đấu thủ, 8 cấu thủ,…

Xuất phát từ lịch thi đấu cho hai đấu thủ ta có thể xây dựng lịch thi đấu cho 4 đấu thủ như sau:

- Lịch thi đấu cho 4 đấu thủ sẽ là một bảng 4 dòng, 3 cột.
- Lịch thi đấu cho 2 đấu thủ 1 và 2 trong ngày thứ 1 chính là lịch thi đấu của hai đấu thủ (bài toán cơ sở). Như vậy ta có Ô(1,1) = “2” và Ô(2,1) = “1”.
- Tương tự ta có lịch thi đấu cho 2 đấu thủ 3 và 4 trong ngày thứ 1, nghĩa là Ô(3,1) =“4” và Ô(4,1) = “3”. (Ta có thể thấy rằng Ô(3,1) = Ô(1,1) + 2 và Ô(4,1) = Ô(2,1) + 2). Bây giờ để hoàn thành lịch thi đấu cho 4 đấu thủ, ta lấy

góc trên bên trái của bảng lắp vào cho góc dưới bên phải và lấy góc dưới bên trái lắp cho góc trên bên phải.

Lịch thi đấu cho 8 đấu thủ là một bảng gồm 8 dòng 7 cột. Góc trên bên trái chính là lịch thi đấu trong 3 ngày đầu của 4 đấu thủ từ 1 đến 4. Các ô của góc dưới bên trái sẽ bằng các ô tương ứng của góc trên bên trái cộng với 4. Đây chính là lịch thi đấu cho 4 đấu thủ 5, 6, 7 và 8 trong 3 ngày đầu. Bây giờ chúng ta hoàn thành việc sắp lịch bằng cách lấp đầy góc dưới bên phải bởi góc trên bên trái và góc trên bên phải bởi góc dưới bên trái.



	1
1	2
2	1

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

	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

Hình 3. Bảng phân lịch thi đấu của 2, 4 và 8 đối thủ.

Dựa vào phân tích trên, hãy viết chương trình sắp lịch thi đấu cho n đấu thủ.

TÀI LIỆU THAM KHẢO

[1] A.V.Aho, J.E. Hopcroft, J.D.Ullman; *Data Structures and Algorithms*; Addison-Wesley; 1983, (Chapter 10).

[2] Jeffrey H Kingston; *Algorithms and Data Structures*; Addison-Wesley; 1998, (Chapter 12).

[3] Đinh Mạnh Tường; *Cấu trúc dữ liệu & Thuật toán*; Nhà xuất bản khoa học và kĩ thuật; Hà nội-2001, (Chương 8).

[4] Nguyễn Đức Nghĩa, Tô Văn Thành; *Toán rời rạc*; 1997, (Chương 3, 5).

[5] Nguyễn Văn Linh, *Giải thuật*, 2003, (Chương 3).

[6] Trang web phân tích giải thuật: http://pauillac.inria.fr/algo/AofA/

[7] Trang web bài giảng về giải thuật:

http://www.cs.pitt.edu/~kirk/algorithmcourses/

[8] Trang tìm kiếm các giải thuật:

http://oopweb.com/Algorithms/Files/Algorithms.html