MỤC LỤC

MỤC LỤC..1
Chương 1: GIỚI THIỆU CHUNG...3
1.1. Thuật toán và cấu trúc dữ liệu:...3
1.2. Một số vấn đề liên quan:...3
1.3. Ngôn ngữ diễn đạt thuật toán:...3
Ngôn ngữ diễn đạt thuật toán được quy ước sử dụng trong giáo trình này là ngôn ngữ tựa C++...3
1.3.1. Cấu trúc của một chương trình chính:...3
1.3.2. Các ký tự:..5
1.3.3. Các câu lệnh:...5
1.3.4. Chương trình con: 6
Chương 2: ThiẾt kẾ và phân tích thuẬt TOÁN...8
2.1. Thiết kế thuật toán:..8
2.1.1. Module hoá thuật toán:..8
2.1.2. Phương pháp tinh chỉnh từng bước:...9
2.2. Phân tích thuật toán: 9
2.2.1. Tính đúng đắn:...9
2.2.2. Mâu thuẫn giữa tính đơn giản và tính hiệu quả:..9
2.2.3. Phân tích thời gian thực hiện thuật toán:..9
Chương 3: đỆ quy (RecursiON)...12
3.1. Đại cương:...12
3.2. Phương pháp để thiết kế một thuật toán đệ quy:..13
3.3. Thuật toán quay lui:...16
Chương 4: MẢng và danh sách tuyẾn tính...18
4.1. Mảng và cấu trúc lưu trữ của mảng:..18
4.2. Danh sách tuyến tính (Linear list):...19
4.3. Ngăn xếp (Stack):...20
4.3.1. Định nghĩa:..20
4.3.2. Lưu trữ Stack bằng mảng:...20
4.3.3. Các ví dụ:...21
4.3.4. Stack với việc cài đặt thuật toán đệ quy:..25
4.4. Hàng đợi (Queue):..28
4.4.1. Định nghĩa:..28
4.4.2. Lưu trữ Queue bằng mảng:..28
Chương 5: danh sách móc nỐi (LINKED LIST)..31
5.1. Danh sách móc nối đơn:...31
5.1.1. Tổ chức danh sách nối đơn:..31
5.1.2. Một số phép toán trên danh sách nối đơn:..31
5.2. Danh sách nối vòng:..33
5.2.1. Nguyên tắc:...33
5.2.2. Thuật toán bổ sung và loại bỏ một nút của danh sách nối vòng:.................34
5.3. Danh sách nối kép:..34
5.3.1. Tổ chức:..34
5.3.2. Một số phép toán trên danh sách nối kép:...35
5.4. Ví dụ về việc sử dụng danh sách móc nối:...36

5.5. Stack và Queue móc nối:...37
Chương 6: CÂY (TREE)..40
6.1. Định nghĩa và các khái niệm:...40
6.1.1. Định nghĩa:...40
6.1.2. Các khái niệm liên quan:...40
6.2. Cây nhị phân:..41
6.2.1. Định nghĩa và tính chất:..41
6.2.2. Biểu diễn cây nhị phân:...42
6.2.3. Phép duyệt cây nhị phân:...43
6.2.4. Cây nhị phân nối vòng:..49
6.3. Cây tổng quát:...51
6.3.1. Biểu diễn cây tổng quát:..51
6.3.2. Phép duyệt cây tổng quát:..53
6.4. Ứng dụng (Biểu diễn cây biểu thức số học):...53
Chương 7: ĐỒ thị (GRAPH)..58
7.1. Định nghĩa và các khái niệm về đồ thị:...58
7.2. Biểu diễn đồ thị:..59
7.2.1. Biểu diễn bằng ma trận lân cận (ma trận kề):...59
7.2.2. Biểu diễn bằng danh sách lân cận (danh sách kề)..59
7.3. Phép duyệt một đồ thị:..61
7.3.1. Tìm kiếm theo chiều sâu:...61
7.3.2.Tìm kiếm theo chiều rộng:...62
7.4. Cây khung và cây khung với giá cực tiểu:...63
Chương 8: SẮP XẾP..65
8.1. Đặt vấn đề:...65
8.2. Một số phương pháp sắp xếp đơn giản:...65
8.2.1. Sắp xếp kiểu lựa chọn:...65
8.2.2. Sắp xếp kiểu chèn:...65
8.2.3. Sắp xếp kiểu nổi bọt:...66
8.3. Sắp xếp kiểu phân đoạn (Sắp xếp nhanh - quick sort):..66
8.4. Sắp xếp kiểu vun đống (Heap sort):..67
8.5. Sắp xếp kiểu trộn (Merge sort):...69
Chương 9: tìm kiẾm...71
9.1. Bài toán tìm kiếm:..71
9.2. Tìm kiếm tuần tự:...71
9.3. Tìm kiếm nhị phân:...71
9.4. Cây nhị phân tìm kiếm:..71
Tài liỆu Tham khẢo...74

CHƯƠNG 1: GIỚI THIỆU CHUNG

1.1. Thuật toán và cấu trúc dữ liệu:

Theo Niklaus Wirth: Thuật toán + Cấu trúc dữ liệu = Chương trình.

Ví dụ: Cho 1 dãy các phần tử, có thể biểu diễn dưới dạng mảng hoặc danh sách.

Cấu trúc dữ liệu và thuật toán có mối quan hệ mật thiết với nhau. do đó việc nghiên cứu các cấu trúc dữ liệu sau này đi đôi với việc xác lập các thuật toán xử lý trên các cấu trúc ấy.

1.2. Một số vấn đề liên quan:

Lựa chọn một cấu trúc dữ liệu thích hợp để tổ chức dữ liệu vào ra và trên cơ sở đó xây dựng được thuật toán xử lý hữu hiệu nhằm đưa tới kết quả mong muốn cho bài toán là một khâu rất quan trọng.

Ta cần phân biệt 2 loại quy cách dữ liệu:

Quy cách biểu diễn hình thức: Còn được gọi là cấu trúc logic của dữ liệu. Đối với mỗi ngôn ngữ lập trình xác định sẽ có một bộ cấu trúc logic của dữ liệu. Dữ liệu thuộc loại cấu trúc nào thì cần phải có mô tả kiểu dữ liệu tương ứng với cấu trúc dữ liệu đó. Ví dụ: Trong C có các kiểu dữ liệu: Struct, Union, File,..

Quy cách lưu trữ: là cách biểu diễn một cấu trúc dữ liệu trong bộ nhớ. Ví dụ: Cấu trúc dữ liệu mảng được lưu trữ trong bộ nhớ theo quy tắc lưu trữ kế tiếp. Có 2 quy cách lưu trữ:

Lưu trữ trong: ví dụ RAM.

Lưu trữ ngoài: ví dụ đĩa (disk).

1.3. Ngôn ngữ diễn đạt thuật toán:

Ngôn ngữ diễn đạt thuật toán được quy ước sử dụng trong giáo trình này là ngôn ngữ tựa C++.

Đặc điểm: Gần giống với Turbo C++, do đó dễ dàng trong việc chuyển một chương trình viết bằng ngôn ngữ tựa C++ sang ngôn ngữ C++.

1.3.1. Cấu trúc của một chương trình chính:

```
void main()
   {
          S1;  \
          S2;   |  Các lệnh của chương trình dùng để diễn tả thuật toán
          ...   |
          Sn;  /
   }
```

Lưu ý:

Để đơn giản, chương trình có thể không cần viết khai báo. Tuy nhiên có thể mô tả trước chương trình bằng ngôn ngữ tự nhiên.
Phần thuyết minh được đặt giữa 2 dấu /* , */ hoặc // để ghi chú trên 1 dòng.

Ví dụ:
```
void main()   /* Chuong trinh chuyen so he 10 thanh he 2*/
{
cout << "n = ";
cin >> n;   /* Nhap n la so he cs 10*/
T=0;
while (n!=0)
  {
  r = n % 2;
  Push(T, r);
  n = n / 2;
  }
cout << "Ket qua chuyen doi sang he co so 2 la: ";
while (T!=0)
  {
  Pop(T, r);
  cout << r;
  }
}
```

1.3.2. Các ký tự:

Các ký tự sử dụng trong chương trình là tương tự như trong C++.
Lưu ý: Trong C++ là có sự phân biệt giữa chữ hoa và chữ thường.

1.3.3. Các câu lệnh:

- *Lệnh gán*: `V = E;`
  Trong đó: V là biến (variable), và E là biểu thức (expression).
  Lưu ý: Có thể dùng phép gán chung. Ví dụ: `a=b=1;`
- *Lệnh ghép*: {S_1; S_2; ...; S_n;} coi như là một câu lệnh (trong đó S_i là các câu lệnh).
- *Lệnh if*: Tương tự như lệnh if của ngôn ngữ C.

```
if (<biểu thức điều kiện>) <câu lệnh>;
```

hoặc:

```
if (<biểu thức điều kiện>) <câu lệnh 1>;
else <câu lệnh 2>;
```

- *Lệnh switch*: Theo cấu trúc sau:

```
switch (<biểu thức>)
{
  case gt1: S1;
  case gt2: S2;
  ...
  case gtn: Sn;
  [default : Sn+1;]
```

```
    }
```

- *Lệnh lặp*: `for, while, do ... while`: Tương tự như các lệnh lặp của C.
- *Lệnh nhảy*: `goto n` (n: số hiệu/nhãn của chương trình).
- *Lệnh vào ra*: `cin` và `cout` giống như C++.

1.3.4. Chương trình con:

```
<kiểu trả về> <Tên hàm>(<danh sách tham số>)
 {
   S1;  ┐
   S2;  │
   ...  │
   S3;  ┘
 [return (giá trị trả về) ]
 }
```

[return (giá trị trả về)] → Báo kết thúc chương trình con

Lưu ý: Nếu hàm có kiểu trả về khác kiểu void thì khi kết thúc hàm phải có câu lệnh **return <giá trị của hàm>** để gán kết quả cho hàm.

Sau đây là ví dụ về hàm có trả về giá trị.

Ví dụ: Viết chương trình con dạng hàm NamNhuan(x). Cho kết quả nếu số x là năm nhuận có giá trị là True(1), ngược lại có giá trị là False(0); chẳng hạn: NamNhuan(1996) cho giá trị 1, NamNhuan(1997) cho giá trị 0. Biết rằng x được gọi là năm nhuận nếu x chia hết cho 4 và x không chia hết cho 100 hoặc x chia hết cho 400.

Cách 1: **`int namnhuan(x)`**

```
  {  if ((x % 4 == 0 && x % 100 != 0)||(x % 400 == 0))
        return 1;
     else
       return 0;
  }
```

Cách 2: **`int  namnhuan(x)`**

```
 { return(((x % 4 == 0) && (x % 100 != 0)) ||
      (x % 400 = 0));
  }
```

Ví dụ viết về chương trình con không có giá trị trả về (hay còn gọi là thủ tục).

Ví dụ: Viết hàm Hoandoi(a, b) để hoán đổi giá trị của 2 biến số a và b cho nhau.

Cách 1:

```
void hoandoi(&a, &b)  //a và b là các tham biến
{  tam=a;
   a=b;
   b=tam;
}
```

Cách 2:

```
void hoandoi(&a, &b)
{  a= a+b;
   b= a-b;
```

```
    a= a-b;
}
```

Lưu ý: Bên trong 1 chương trình con có thể dùng lệnh return; (thoát khỏi chương trình con), `exit(1)` (thoát khỏi chương trình chính).

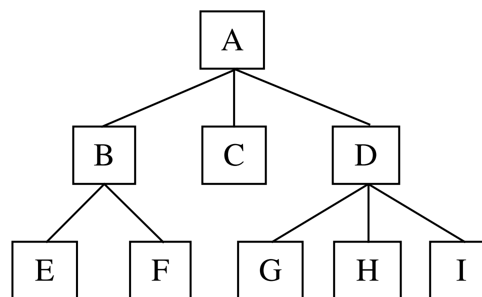
CHƯƠNG 2: THIẾT KẾ VÀ PHÂN TÍCH THUẬT TOÁN

2.1. Thiết kế thuật toán:

2.1.1. Module hoá thuật toán:

Các bài toán ngày càng đa dạng và phức tạp, do đó thuật toán mà ta đề xuất càng có quy mô lớn và việc viết chương trình cần có một lượng lập trình đông đảo. Muốn làm được việc này , người ta phân chia các bài toán lớn thành các bài toán nhỏ (module). Và dĩ nhiên một module có thể chia nhỏ thành các module con khác nữa,... bấy giờ việc tổ chức lời giải sẽ được thể hiện theo một cấu trúc phân cấp.

Ví dụ:



Quá trình module hoá bài toán được xem là nguyên lý "chia để trị" (divide & conquer) hay còn gọi là thiết kế từ đỉnh xuống (top-down) hoặc là thiết kế từ khái quát đến chi tiết (specialization).

Việc module hoá trong lập trình thể hiện ở:

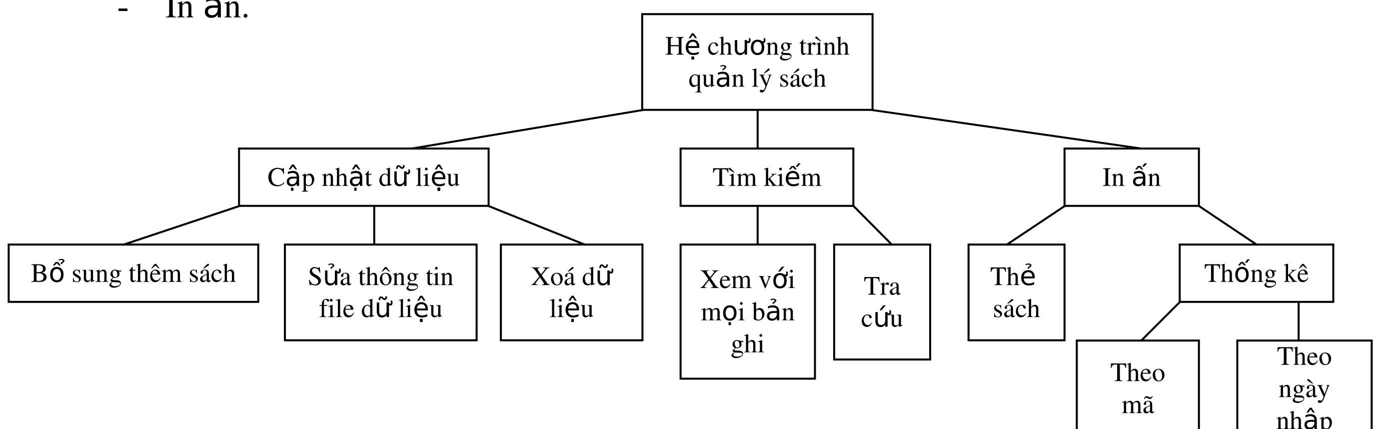
Các chương trình con.

Cụm các chương trình con xung quanh một cấu trúc dữ liệu nào đó. Chẳng hạn, thư viện trong C.

Ví dụ: Chương trình quản lý đầu sách của một thư viện nhằm phục vụ độc giả tra cứu sách. Cụ thể, giả sử ta đã có một file dữ liệu gồm các bảng ghi về các thông tin liên quan đến một đầu sách như: tên sách, mã số, tác giả, nhà xuất bản, năm xuất bản, giá tiền, ...

Yêu cầu:

- Cập nhật dữ liệu được.
- Tìm kiếm.
- In ấn.



Nhận xét:

- Việc module hoá làm cho bài toán được định hướng rõ ràng.
- Bằng cách này, người ta có thể phân chia công việc cho đội ngũ lập trình.
- Đây là một công việc mất nhiều thời gian.

2.1.2. Phương pháp tinh chỉnh từng bước:

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

Xuất phát từ ngôn ngữ tự nhiên của thuật toán, thuật toán sẽ được chi tiết hoá dần dần và cuối cùng công việc xử lý sẽ được thay thế dần bởi các câu lệnh (của một ngôn ngữ lập trình nào đó). Quá trình này là để trả lời dần dần các câu hỏi: What? (làm gì?), How (làm như thế nào?)

2.2. Phân tích thuật toán:

Chất lượng của một chương trình hay thuật toán bao gồm:

- Tính đúng đắn.
- Tính đơn giản (dễ hiểu, dễ quản lý, dễ lập).
- Tính tối ưu (hiệu quả) về mặt thời gian cũng như không gian nhớ.

2.2.1. Tính đúng đắn:

Đây là một yêu cầu phân tích quan trọng nhất cho một thuật toán. Thông thường, người ta thử nghiệm (test) nhờ một số bộ dữ liệu nào đó để cho chạy chương trình rồi so sánh kết quả thử nghiệm với kết quả mà ta đã biết. Tuy nhiên, theo Dijkstra: "Việc thử nghiệm chương trình chỉ chứng minh sự có mặt của lỗi chứ không chứng minh sự vắng mặt của lỗi".

Ngày nay, với các công cụ toán học người ta có thể chứng minh tính đúng đắn của một thuật toán.

2.2.2. Mâu thuẫn giữa tính đơn giản và tính hiệu quả:

Một thuật toán đơn giản (dễ hiểu) chưa hẳn tối ưu về thời gian và bộ nhớ. Đối với những chương trình chỉ dùng một vài lần thì tính đơn giản có thể coi trọng nhưng nếu chương trình được sử dụng nhiều lần (ví dụ, các phần mềm) thì thời gian thực hiện rõ ràng phải được chú ý.

Yêu cầu về thời gian và không gian ít khi có một giải pháp trọn vẹn.

2.2.3. Phân tích thời gian thực hiện thuật toán:

Thời gian thực hiện thuật toán phụ thuộc vào nhiều yếu tố:

- Kích thước dữ liệu đưa vào (dung lượng). Nếu gọi n là kích thước dữ liệu vào thì thời gian thực hiện một thuật toán, ký hiệu là T(n).
  - Tốc độ xử lý của máy tính, bộ nhớ (RAM).
  - Ngôn ngữ để viết chương trình.

Tuy nhiên, ta có thể so sánh thời gian thực hiện của hai thuật toán khác nhau.

Ví dụ: Nếu thời gian thực hiện của thuật toán thứ nhất $T_1(n) = Cn^2$ (C: hằng dương) và thời gian thực hiện thuật toán thứ hai $T_2(n) = Kn$ (K: hằng) thì khi n khá lớn, thời gian thực hiện thuật toán 2 sẽ tối ưu hơn so với thuật toán 1.

Cách đánh giá thời gian thực hiện thuật toán theo kiểu trên được gọi là đánh giá thời gian thực hiện thuật toán theo "độ phức tạp tính toán của thuật toán".

2.2.3.1. Độ phức tạp tính toán của thuật toán:

Nếu thời gian thực hiện một thuật toán là $T(n) = Cn^2$ (C: hằng), thì ta nói rằng: Độ phức tạp tính toán của thuật toán này có cấp là n^2 và ta ký hiệu $T(n) = O(n^2)$.

Tổng quát: $T(n) = O(g(n))$ thì ta nói độ phức tạp của thuật toán có cấp là g(n).

2.2.3.2. Xác định độ phức tạp của thuật toán:

Việc xác định độ phức tạp tính toán của một thuật toán nói chung là phức tạp. Tuy nhiên, trong thực tế độ phức tạp của một thuật toán có thể được xác định từ độ phức tạp từng phần của thuật toán. Cụ thể, ta có một số quy tắc sau:

- *Quy tắc tính tổng*:

Nếu chương trình P được phân tích thành 2 phần: P_1, P_2 và nếu độ phức tạp của P_1 là $T_1(n) = O(g_1(n))$ và độ phức tạp của P_2 là $T_2(n) = O(g_2(n))$ thì độ phức tạp của P là: $T(n) = O(\max(g_1(n), g_2(n)))$.

Ví dụ: $g_1(n) = n^2$, $g_2(n) = n^3$. Suy ra: $T(n) = O(n^3)$.

Lưu ý: $g_1(n)$ $g_2(n)$ (n n_0) $O(g_1(n) + g_2(n)) = O(g_2(n))$

Ví dụ: $O(n + \log_2 n) = O(n)$

- *Quy tắc nhân:*

Nếu độ phức tạp của P_1 là $O(g_1(n))$, độ phức tạp của P_2 là $O(g_2(n))$ thì độ phức tạp của P_1 lồng P_2 (P_1 câu lệnh lặp) thì độ phức tạp tính toán là $O(g_1(n).g_2(n))$.

Lưu ý:

- Câu lệnh gán, cin, cout, if, switch có thời gian thực hiện bằng hằng số C = O(1).
- Câu lệnh lặp trong vòng g(n) lần thì sẽ có thời gian thực hiện là O(g(n)).
- O(Cg(n)) = O(g(n)) (C: hằng)

Ví dụ:

1) Câu lệnh:
```
for (i=1;i<=n;i++)          // O(n)
    P=P*i;                  // O(1)
```
có thời gian thực hiện là: O(n*1) = O(n).

2)
```
for (i=1;i<=n;i++)
```

```
for (j=1;j<=n;j++) x=x+1;
```

có thời gian thực hiện là: $O(n*n*1) = O(n^2)$.

- Thông thường, để xác định độ phức tạp tính toán của một thuật toán, người ta đi tìm một lệnh/phép toán có số lần thực hiện là nhiều nhất (lệnh/phép toán tích cực) từ đó tính số lần này   độ phức tạp của tính toán.
- Có khi thời gian thực hiện một thuật toán còn phụ thuộc vào đặc điểm của dữ liệu. Bấy giờ T(n) trong trường hợp thuận lợi nhất có thể khác T(n) trong trường hợp xấu nhất. Tuy nhiên, thông thường người ta vẫn đánh giá độ phức tạp tính toán của thuật toán thông qua T(n) trong trường hợp xấu nhất.

  <u>Ví dụ</u>: Cho một dãy gồm có n phần tử mảng: V[0], V[1], ..., V[n-1]. X là một giá trị cho trước.

```
void timkiem(x)
{
  found=0;// gán giá trị cho found lúc ban đầu là False
  i=0;
  while ((i< n) && (!found))
    if (v[i]==x)
      {
        cout << i; found=1;
      }
    else i=i+1;
  if (found==0)
    cout << “không có”;
}
```

T(n) thuận lợi = O(1)   (X = V[0])
T(n) xấu nhất = O(n)   (X   V[i],   i=0..n-1)

T(n) = O(n)

CHƯƠNG 3: ĐỆ QUY (RECURSION)

3.1. Đại cương:

- Chương trình đệ quy là chương trình gọi đến chính nó.

  Ví dụ: Một hàm đệ quy là một hàm được định nghĩa dựa vào chính nó.

- Trong lý thuyết tin học, người ta thường dùng thủ thuật đệ quy để định nghĩa các đối tượng.

  Ví dụ: Tên biến được định nghĩa như sau:

  - Mỗi chữ cái là một tên.
  - Nếu t là tên biến thì t <chữ cái>, t <chữ số> cũng là tên biến.

- Một chương trình đệ quy hoặc một định nghĩa đệ quy thì không thể gọi đến chính nó mãi mãi mà phải có một điểm dừng đến một trường hợp đặc biệt nào đó, mà ta gọi là trường hợp suy biến (degenerate case).

  Ví dụ: Cho số tự nhiên n, ta định nghĩa n! như sau: $n! = \begin{cases} n * (n-1)! \\ 0! \quad 1 \end{cases}$

- Lời giải đệ quy: Nếu lời giải của một bài toán T nào đó được thực hiện bằng một lời giải của bài toán T' có dạng giống như T, nhưng theo một nghĩa nào đó T' là "nhỏ hơn" T và T' có khuynh hướng ngày càng tiếp cận với trường hợp suy biến.

  Ví dụ: Cho dãy các phần tử mảng V[1], V[2], ..., V[n] đã được sắp xếp theo thứ tự tăng dần, gọi X là một giá trị bất kỳ. Viết thuật toán tìm kiếm để in vị trí của phần tử nào đó trong mảng có giá trị bằng X (nếu có). Ngược lại, thông báo không có.

```
void timkiem(d, c, x)
{
  if (d>c)
      cout << “khong co”;
  else
    {
      g=(d+c)/ 2;
      if (x==V[g])
             cout << g;
      else if (x<V[g])  timkiem(d, g-1, x);
            else timkiem(g+1, c, x);
    }
}
```

Nhận xét:

Bài toán tìm kiếm ban đầu được tách thành các bài toán tìm kiếm với phạm vi nhỏ hơn cho đến khi gặp phải các trường hợp suy biến. Chính

việc phân tích đó, người ta đã xem thuật toán đệ quy là thuật toán thể hiện phương pháp "chia để trị".

Nếu thủ tục hoặc hàm chứa lời gọi đến chính nó (ví dụ trên) thì được gọi là đệ quy trực tiếp. Ngược lại, có thủ tục chứa lời gọi đến thủ tục khác mà ở thủ tục này chứa lại lời gọi đến nó thì được gọi là đệ quy gián tiếp, hay còn gọi là đệ quy tương hỗ hay còn gọi là Forward.

Ví dụ:

```
void Ba(int n)
{
   cout << n;
   if (n>0)  Ong(n-1);
}

void Ong(int n);
{
   cout << n;
   if (n>0)  Ba(n-1);
}
 void main()
{
   Ong(3);
}
```

Kết quả:

```
3 Ong
2 Ba
1 Ong
0 Ba
```

3.2. Phương pháp để thiết kế một thuật toán đệ quy:

- Tham số hoá bài toán.
- Phân tích trường hợp chung (đưa bài toán dưới dạng bài toán cùng loại nhưng có phạm vi giải quyết nhỏ hơn theo nghiã dần dần sẽ tiến đến trường hợp suy biến).
- Tìm trường hợp suy biến.

Ví dụ:

1) *Lập hàm GT(n) = n!*

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

2) *Dãy số Fibonaci*: $F_1 = F_2 = 1$;

$F_n = F_{n-1} + F_{n-2.}$ (n 3)

```
long F(n)
{  if (n  2)  return 1;
```

```
        else return F(n-1)+F(n-2);
    }
```

Nhận xét:

- Thông thường thay vì sử dụng lời giải đệ quy cho một bài toán, ta có thể thay thế bằng lời giải không đệ quy (khử đệ quy) bằng phương pháp lặp.

- Việc sử dụng thuật toán đệ quy có:

Ưu điểm	Khuyết điểm
Thuận lợi cho việc biểu diễn bài toán. Gọn (đối với chương trình).	Có khi không được tối ưu về thời gian. Có thể gây tốn bộ nhớ → xảy ra hiện tượng tràn bộ nhớ ngăn xếp (Stack) nếu dữ liệu lớn.

- Chính vì vậy, trong lập trình người ta cố tránh sử dụng thủ tục đệ quy nếu thấy không cần thiết.

Bài tập:

1) Viết hàm luỹ thừa float `lt(float x, int n)` cho ra giá trị x^n.

2) Viết chương trình nhập vào số nguyên rồi đảo ngược số đó lại (không được dùng phương pháp chuyển số thành xâu).

3) Viết chương trình cho phép sản sinh và hiển thị tất cả các số dạng nhị phân độ dài n (có gồm n chữ số).

Ví dụ 1: Viết thủ tục in xâu đảo ngược của xâu X.

Trước khi xây dựng hàm InNguoc thì ta xây dựng hàm tách chuỗi con từ chuỗi mẹ trước từ vị trí là *batdau* và lấy *soluong* ký tự.

```
char *copy(char *chuoi,int batdau,int soluong)
 { int i; char *tam;
 tam=(char *)malloc(100);
 for(i=(batdau-1);i<strlen(chuoi)&& i<(batdau-1+soluong);i++)
     tam[i-(batdau-1)]=chuoi[i];
 tam[i]=NULL;
 return tam;
}
```

Cách 1:

- Trường hợp chung: + In ký tự cuối của xâu X.

+ Đảo ngược phần còn lại.

- Trường hợp suy biến: Nếu xâu rỗng thì không làm gì hết.

```
void InNguoc(X){
     if (X[0] !='')
        {
          cout << X[strlen(X)-1];
          InNguoc(copy(X,0,strlen(x)-2);
        }
  }
```

Cách 2:

- Trường hợp chung: + Đảo ngược xâu X đã bỏ ký tự đầu tiên.

  + In ký tự đầu tiên của X.

- Trường hợp suy biến: Nếu xâu rỗng thì không làm gì hết.

```
void Innguoc(X){
  if (X!="")
    {
      InNguoc(copy(X, 1,strlen(X)-2);
      cout << X[0];
    }
  }
```

Ví dụ 2: Bài toán tháp Hà nội: Cho ba cọc A, B, C; có n đĩa khác nhau được xếp theo thứ tự nhỏ trên lớn dưới nằm trên cọc A. Yêu cầu: Chuyển chồng đĩa từ cọc A sang cọc C với điều kiện:

- Mỗi lần chỉ được chuyển một đĩa.
- Không có trường hợp đĩa lớn được đặt trên đĩa nhỏ.
- Có thể dùng cọc B làm cọc trung gian.

➢ Tham số hoá bài toán: `HaNoi(n, A, B, C)   //char A, B, C`

Trong đó: n: Số đĩa.
A: Cọc nguồn cần chuyển đĩa đi.
B: Cọc trung gian.
C: Cọc đích để chuyển đĩa đến.

Chương trình chính như sau:

```
void main()
  {
   cin >> n;
   A= 'A';  B= 'B';  C= 'C';
   HaNoi(3, A, B, C);
  }
```

➢ Thuật toán đệ quy:

- Trường hợp suy biến:

Nếu n = 1 thì chuyển đĩa từ cọc A qua cọc C

- Trường hợp chung (n 2):

Thử với n=2: + Chuyển đĩa thứ nhất từ A sang B.
+ Chuyển đĩa thứ 2 từ A sang C.
+ Chuyển đĩa thứ nhất từ B sang C.

→ Tổng quát: + Chuyển (n -1) đĩa từ A sang B (C làm trung gian).
+ Chuyển 1 đĩa từ A sang C (B: trung gian)
+ Chuyển (n -1) đĩa từ B sang C (A: trung gian).

Suy ra thuật toán đệ quy:

```
void HaNoi(n, A, B, C)
{
  if (n==1)  cout << A << "→" << C;
  else
    {
      HaNoi(n -1, A, C, B);
      HaNoi(1, A, B, C);
      HaNoi(n -1, B, A, C);
}   }
```

3.3. Thuật toán quay lui:

Ta có thể dùng kỹ thuật đệ quy để diễn tả thuật toán quay lui. Bài toán sử dụng thuật toán quay lui thường có dạng: Xác định một bộ gồm n thành phần: x_1, x_2,..., x_n thoả mãn điều kiện B nào đó.

Phương pháp của thuật toán quay lui:

- Giả sử ta đã xác định được i-1 thành phần: x_1, x_2,..., x_{i-1}. Để xác định thành phần x_i, ta duyệt tất cả các khả năng có thể có của nó.

  <u>*Ví dụ*</u>: x_i có thể có giá trị từ 1 đến 8; gọi j là các giá trị có thể có của x_i, lúc đó ta dùng câu lệnh For như sau: `For (j=1;j<8;j++) ...`

- Bây giờ, với mỗi khả năng j ta luôn kiểm tra xem j có được chấp nhận không? (liệu bộ (x_1, x_2, …, x_i) hiện tại có thoã mãn điều kiện B hay không?)

➢ Như vậy, xảy ra 2 trường hợp:

Nếu chấp nhận j:

- Xác định x_i theo j: `xi=j;`
- Sau đó, nếu i còn nhỏ hơn n thì ta tiến hành xác định x_{i+1}.
- Ngược lại (i = n) thì ta được một lời giải.
- Kiểm tra j tiếp theo.

Nếu tất cả các khả năng của j không có khả năng nào được chấp nhận thì quay lại bước trước để xác định lại x_{i-1}. (Cơ chế hoạt động trong bộ nhớ của thuật toán đệ quy giúp có thể thực hiện được điều này).

➢ Việc xác định x_i có thể mô tả qua thủ tục đệ quy sau:

```
void Try(i) //Thử xem xi sẽ nhận giá trị nào
  for (mỗi khả năng j của xi)
    {
      if <Chấp nhận>
        {
          <Xác định xi theo j>;  // Ví dụ: x[i]=j;
          if (i==n)  <Ghi nhận một lời giải>;
            else Try(i+1);
        }
    }
```

Bài tập:

1) Tìm tất cả các hoán vị của một mảng gồm có n phần tử.

2) Bài toán 8 con hậu: Hãy tìm cách đặt 8 quân hậu trên một bàn cờ vua sao cho không có quân hậu nào có thể ăn các quân hậu khác.

CHƯƠNG 4: MẢNG VÀ DANH SÁCH TUYẾN TÍNH

4.1. Mảng và cấu trúc lưu trữ của mảng:

- Mảng là cấu trúc dữ liệu đơn giản và thông dụng trong nhiều ngôn ngữ lập trình.
- Mảng là một tập có thứ tự gồm một số cố định các phần tử có cùng quy cách.

  Ví dụ: Trong C, để khai báo một dãy số nguyên n phần tử: a[0], a[1],..., a[n-1] (với n   100), ta khai báo mảng a như sau:

  int a[100] ;

  Lúc này, việc truy xuất sẽ thông qua các phần tử của mảng, ký hiệu: a[0], a[1],...a[99].
- Ma trận là một mảng 2 chiều.

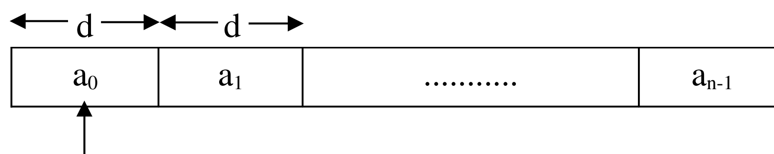
  Ví dụ: float B[100][100];

  Khi đó, B[i][j] là một phần tử của ma trận B. Trong đó i là hàng còn j là cột.
- Tương tự ta cũng có mảng 3 chiều, mảng 4 chiều.

❖ *Cấu trúc lưu trữ:*

Cách lưu trữ mảng thông thường (đối với mọi ngôn ngữ lập trình) là lưu trữ theo kiểu kế tiếp.

Ví dụ: Gọi a là mảng 1 chiều gồm có n phần tử, mỗi phần tử có độ dài là d (chiếm d byte) và được lưu trữ kế tiếp như hình dưới đây:



Loc (a_0): địa chỉ phần tử a_0

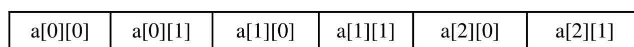
→ địa chỉ của phần tử thứ a_i:

$$Loc (a_i) = Loc (a_0) + d*i$$

Lưu ý:

- Đối với mảng nhiều chiều, việc tổ chức lưu trữ cũng được thực hiện tương tự:

Ví dụ: `int a[3][2];`



→ Địa chỉ của phần tử a_{ij}:

$$Loc (a[i][j]) = Loc (a[0][0]) + d*(i*n + j)$$

Trong đó, n là số cột của ma trận.

Bài tập:

1) Viết công thức tổng quát để tính địa chỉ của một phần tử nào đó của một mảng n chiều (Loc a[i_0, …, i_{n-1}]), với chỉ số các chiều này lần lượt là: $b_0..b'_0$, $b_1..b'_1$,....,

$b_{n-1}..b'_{n-1}$; trong đó: i_0 [$b_0..b'_0$], i_1 [$b_1..b'_1$], …, i_{n-1} [$b_{n-1}..b'_{n-1}$]. Địa chỉ này phụ thuộc vào địa chỉ của chỉ số đầu tiên a[b_0, b_1,.., b_{n-1}]. Cho d là độ dài của một phần tử.

Lưu ý: do các phần tử của mảng thường được lưu trữ kế tiếp nhau nên việc truy nhập vào chúng nhanh, đồng đều với mọi phần tử (ưu điểm). Trong lúc đó, nhược điểm của việc lưu trữ mảng là:

+ Phải khai báo chỉ số tối đa, do đó có trường hợp gây lãng phí bộ nhớ.
+ Khó khăn trong việc thực hiện phép xoá / chèn một phần tử trong mảng.

2) Giả sử trong bộ nhớ có mảng a gồm n phần tử a_0, a_1, ... ,a_{n-1}.

Hãy viết các hàm sau:

+ void Xoa(i): Xoá phần tử thứ i trong mảng này.
+ void ChenSau(i, x): Chèn sau phần tử thứ i một phần tử có giá trị là x.

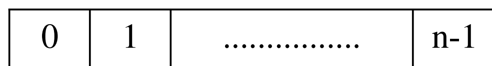
4.2. Danh sách tuyến tính (Linear list):

❖ *Định nghĩa:*

Danh sách tuyến tính là một dãy có thứ tự a_1, a_2,..., a_n (n>=0). Nếu n=0 được gọi là danh sách rỗng. Ngược lại: a_1 được gọi là phần tử đầu tiên, a_n được gọi là phần tử cuối cùng, và n được gọi là chiều dài của danh sách.

- Đối với danh sách tuyến tính, với mỗi phần tử a_i (i =1, n-1) thì có phần tử tiếp theo là a_{i+1} và với mỗi phần tử a_i (i = 2..n) thì có phần tử đứng trước là a_{i-1}.
- Danh sách tuyến tính khác cơ bản với mảng một chiều ở chỗ là kích thước của danh sách không cố định bởi vì phép bổ sung và phép loại bỏ thường xuyên tác động lên một danh sách. Ví dụ: Stack.
- Có nhiều cách để lưu trữ một danh sách tuyến tính:
  + Lưu trữ theo địa chỉ kế tiếp bằng mảng 1 chiều.
  + Lưu trữ địa chỉ bằng con trỏ (sử dụng danh sách móc nối).
  + Lưu trữ ra file (sử dụng bộ nhớ ngoài).
- Với danh sách tuyến tính, ngoài phép bổ sung và loại bỏ còn có một số phép sau:

+ Phép ghép 2 hoặc nhiều danh sách thành một danh sách (xem như bài tập, làm trên mảng và trỏ).



+ Phép tách (tách một danh sách thành 2 danh sách).
+ Sao chép một danh sách ra nhiều danh sách (2 danh sách).
+ Cập nhật hoặc sửa đổi nội dung các phần tử của danh sách.
+ Sắp xếp các phần tử trong danh sách theo thứ tự ấn định trước.
+ Tìm kiếm một phần tử trong danh sách thoả mãn một điều kiện cho trước.

4.3. Ngăn xếp (Stack):

4.3.1. Định nghĩa:

Stack là một kiểu danh sách tuyến tính đặc biệt, trong đó phép bổ sung và loại bỏ chỉ thực hiện ở một đầu gọi là đỉnh Stack (đầu kia gọi là đáy của Stack).

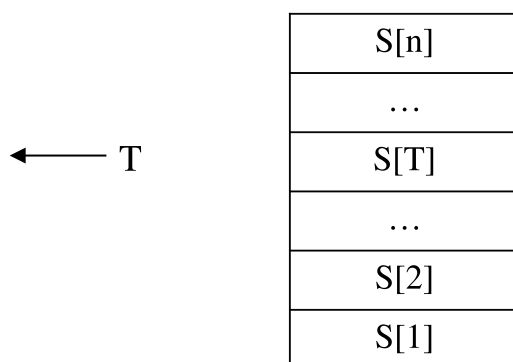
Nguyên tắc bổ sung và loại bỏ đối với Stack được gọi là nguyên tắc vào sau ra trước (LIFO – Last In First Out).

4.3.2. Lưu trữ Stack bằng mảng:

Vì Stack là một danh sách tuyến tính nên có thể sử dụng mảng một chiều để tổ chức một Stack. Chẳng hạn: sử dụng mảng S để lưu trữ dãy các phần tử: S[1], S[2],..., S[n] (n gọi là số phần tử cực đại của mảng S).

Gọi T là chỉ số của phần tử đỉnh của Stack. T được sử dụng để theo dõi vị trí đỉnh của Stack nên nếu sử dụng danh sách móc nối để tổ chức một Stack thì T được xem như là một con trỏ chỉ vào vị trí đỉnh của Stack.

Giá trị của T sẽ tăng lên một đơn vị khi bổ sung một phần tử vào danh sách và sẽ giảm bớt 1 khi loại bỏ một phần tử ra khỏi Stack.



Lưu ý:

- Khi T = n thì không thể bổ sung thêm (hay nói cách khác là Stack đầy).
- Khi T = 0 thì không thể loại bỏ phần tử vì khi đó Stack rỗng (hay Stack cạn).

➢ *Thuật toán bổ sung một phần tử X vào Stack S có đỉnh là T:*