

TRƯỜNG ĐẠI HỌC GIAO THÔNG VẬN TẢI  
KHOA CÔNG NGHỆ THÔNG TIN

BÀI GIẢNG

PHÂN TÍCH THIẾT KẾ THUẬT TOÁN

Phạm Xuân Tích

## **Mục Lục**

Mục Lục .....	2
Chương 1. Tổng quan thuật toán .....	4
1.1. Thuật toán và đặc trưng.....	4
1.1.1. Tính xác định .....	5
1.1.2. Tính tuần tự các bước .....	6
1.1.3. Tính dừng.....	8
1.1.4. Tính đa dạng .....	9
1.1.5. Tính phổ dụng.....	13
1.1.6. Tính hình thức hóa.....	15
1.2. Biểu diễn thuật toán .....	16
1.2.1. Biểu diễn đặc tả tự nhiên .....	16
1.2.2. Biểu diễn bằng sơ đồ khối .....	17
1.2.3. Biểu diễn bằng tựa ngôn ngữ lập trình Pascal .....	19
1.3. Đánh giá thuật toán .....	22
1.3.1. Khái niệm.....	22
1.3.2. Đánh giá độ phức tạp thời gian.....	23
1.3.3. Các kí hiệu tiệm cận: $O, \Omega, \Theta$ .....	25
1.3.4. Đánh giá độ phức tạp của thuật toán .....	27
1.4. Bài tập .....	35
Chương 2. Phân tích thiết kế thuật toán lập và đệ quy .....	36
2.1. Phân tích thiết kế thuật toán.....	37
2.1.1. Các yếu tố phân tích, thiết kế thuật toán.....	37
2.1.2. Một số ví dụ .....	38
2.2. Phân tích thiết kế thuật toán tính tổng .....	44
2.3. Phân tích thiết kế thuật toán trên dãy.....	49
2.3.1. Tìm phần tử âm lớn nhất trong dãy .....	49
2.3.2. Đếm số đoạn cắt trực hoành.....	54
2.4. Bài tập .....	59

Chương 3. Phân tích thiết kế chia để trị .....	61
3.1.    Phương pháp chia để trị .....	61
3.2.    Một số thuật toán giảm để trị .....	66
3.2.1.    Bài toán tìm ước chung lớn nhất của hai số .....	67
3.2.2.    Bài toán xóa chữ số.....	68
3.2.3.    Bài toán tính lũy thừa.....	70
3.2.4.    Bài toán tìm số Fibonacci thứ n .....	71
3.2.5.    Bài toán tìm số lớn thứ k trong dãy n phần tử .....	73
3.2.6.    Bài toán Robot cắt thanh kim loại .....	76
3.3.    Một số thuật toán chia để trị.....	80
3.3.1.    Bài toán leo thang .....	80
3.3.2.    Bài toán mọi con đường về không.....	82
3.3.3.    Bài toán tìm dãy con liên tục có tổng lớn nhất.....	85
3.3.4.    Bài toán đếm số nghịch thế.....	88
3.3.5.    Bài toán đổi tiền.....	92
3.4.    Thuật toán chia để trị trên dãy số.....	95
3.4.1.    Thuật toán giảm để trị giải quyết bài toán .....	96
3.4.2.    Thuật toán chia để trị giải quyết bài toán .....	98
3.5.    Bài tập .....	100
Chương 4. Phân tích thiết kế quay lui .....	102

## **Chương 1. Tổng quan thuật toán**

### **1.1. Thuật toán và đặc trưng**

#### ***Khái niệm***

Thuật toán hay giải thuật là một quy trình gồm một dãy các thao tác biến đổi từ dữ liệu vào với rất nhiều các ràng buộc cụ thể thành dữ liệu đầu ra sau hữu hạn bước.

Một chương trình (thủ tục hoặc hàm) là một cách biểu diễn hình thức thuật toán bởi một ngôn ngữ lập trình cụ thể.

#### ***Ví dụ: Thuật toán Euclid tìm ước chung lớn nhất của hai số nguyên a, b***

Theo nhà toán học Hy Lạp cổ đại Euclid thì ước chung lớn nhất của hai số nguyên a, b (ký hiệu  $(a,b)$ ) bằng ước chung lớn nhất của b với phần dư a chia cho b ( $(a,b) = (b, a \bmod b)$ ), do đó ta dùng phép lặp biến đổi tới khi b bằng 0, thu được ước chung lớn nhất là a.

#### ***Chương trình:***

```
#include<stdio.h>
#define input(n) printf("Nhap so %s : ",#n);
scanf("%u",&n);
int Euclid(int a, int b){
    while(b)
    {
        int r=a%b;
        a=b;
        b=r;
    }
    return a;
}
int main()
{
    int x,y;
    input(x); input(y);
    printf("Uoc chung lon nhat la %d",Euclid(x,y));
}
```

**Minh họa:** Ta muốn tìm ước chung lớn nhất của  $a = 300$  và  $b = 1980$  theo dãy biến đổi sau:

$$(300, 1980) = (1980, 300) = (300, 180) = (180, 120) = (120, 60) = (60, 0) = 60.$$

### 1.1.1. Tính xác định

Mỗi thuật toán được gắn với một bài toán cụ thể được xác định đầu vào và đầu ra và điều kiện của chúng.

**Ví dụ 1.** Nhập vào các số thực  $a, b, c$  biện luận về nghiệm của phương trình  $ax^2 + bx + c = 0$ .

**Phân tích thuật toán:**

- Nếu  $a = 0$  phương trình suy biến về  $bx + c = 0$
- Nếu  $a$  khác 0 ta đặt  $b = b/2$  đưa về phương trình  $ax^2 + 2bx + c = 0$  để dàng biện luận phương trình này.

**Chương trình:**

```
#include<math.h>
int main(){
    float a,b,c,d;
    scanf("%f%f%f",&a,&b,&c);
    if(a==0){
        if(b==0) printf("vo nghiem":"vo so nghiem");
        else printf("%.3f",-c/b);
    }
    else
    {
        b/=2;
        d=b*b-a*c;
        if(d<0) printf("vo nghiem");
        else if(d==0) printf("%.2f",-b/a);
        else
        {
            d=sqrt(d);
            printf("%.3f\n%.3f",(-b-d)/a,(-b+d)/a);
        }
    }
}
```

**Ví dụ 2.** Nhập vào các số phức a, b, c biến luận về nghiệm phức của phương trình  $ax^2 + bx + c = 0$ .

**Chương trình:**

```
#include<bits/stdc++.h>
using namespace std;
typedef complex<float> SP;
int main(){
    SP a,b,c,x1,x2, k(0,0);
    cout<<"a(real,img) : "; cin>>a;
    cout<<"b(real,img) : "; cin>>b;
    cout<<"c(real,img) : "; cin>>c;
    if(a==k)
    {
        if(b==k) cout<<(c==k?"vo so nghiem":"vo nhien");
        else cout<<"Nghiem : "<<-c/b;
    }
    else
    {
        b=b/SP(2,0);
        SP d=sqrt(b*b-a*c);
        x1=(-b-d)/a;
        x2=(-b+d)/a;
        cout<<"X1 : "<<x1<<"\nX2 : "<<x2;
    }
}
```

### 1.1.2. Tính tuần tự các bước

Mỗi thuật toán đều tuân thủ các bước theo trình tự nhất định, bước nào trước thực hiện trước, bước nào sau thực hiện sau.

**Ví dụ:** Thuật toán Eratosthenes sàng những số nguyên tố nhỏ hơn số nguyên dương n.

Nhà toán học Hy Lạp cổ đại Eratosthenes muốn tìm các số nguyên tố thuộc  $[2,n]$ , ông đưa tất cả các số lên sàng và lọc dần loại bỏ những số chia hết

cho 2, chia hết cho 3, ... đến lúc không còn số nào chia hết cho số khác ta được danh sách các số nguyên tố.

**Bước 1.** Dùng một mảng s đánh dấu từ 2 đến n đều là true.

**Bước 2.** Duyệt lần lượt từ 2 đến n số nào loại thì chuyển thành false.

Xét 2 là số nguyên tố, loại các bội của 2 nhỏ hơn hoặc bằng n: 4,6,8...

Xét 3 là số nguyên tố, loại các bội của 3 nhỏ hơn hoặc bằng n: 9, 12,...

Xét 4 đã bị loại không quan tâm vì mọi bội của 4 đều là bội của 2

Xét i bất kỳ ( $2 \leq i \leq n$ ) nếu i đã bị loại thì không quan tâm ngược lại nó là số nguyên tố ta tiến hành loại các bội của i từ  $i*i$  trở đi vì nếu nhỏ hơn đã bị số khác loại. Chẳng hạn nếu  $i = 11$  thì số 22, 44, 66, 88, 110 đã bị 2 loại, còn 33, 55, 77 lần lượt bị loại bởi 3, 5, 7, nên chúng ta chỉ cần loại bội của 11 từ 121 trở đi đến hết những số nhỏ hơn hay bằng n.

**Bước 3.** Sau khi kết thúc ta thu được mảng s những vị trí true là số nguyên tố

**Chương trình:** áp dụng thuật toán in ra những số nguyên tố nhỏ hơn 1000.

```
#include<bits/stdc++.h>
using namespace std;
void Eratosthenes(int n){
    bool s[n+5];
    for(int i=2; i<=n; s[i++]=1);
    for(int i=2; i<=n; i++)
        if(s[i]){
            cout<<i<<"\t";
            for(int j=i*i; j<=n; j+=i) s[j]=0;
        }
}
int main(){
    Eratosthenes (1000); //in cac so nguyen to nho hon 1000
}
```

**1.1.3. Tính dừng**

Thuật toán chỉ có ý nghĩa nếu nó dừng sau hữu hạn bước, còn chạy vô hạn không dừng không thu được kết quả.

**Ví dụ:** Tìm nghiệm gần đúng của phương trình  $f(x) = 0$  bằng phương pháp lặp đơn trên một khoảng  $[a, b]$  và sai số  $\varepsilon$  cho trước.

Theo nguyên lý ánh xạ co trong không gian Banach, chúng ta biến đổi phương trình  $f(x) = 0$  về dạng thuận lợi cho phép lặp  $x = g(x)$ , nếu thỏa mãn tính chất  $g(x)$  là một hàm khả vi liên tục trên đoạn  $[a, b]$  và có đạo hàm  $|g'(x)| \leq q < 1$  với  $\forall x \in [a, b]$ . Khi đó ta khởi gán  $x_0$  một giá trị bất kỳ trên đoạn  $[a, b]$  và thực hiện bước lặp  $x_{k+1} = g(x_k)$  thì dần dần  $x_k$  sẽ hội tụ về nghiệm  $x^*$  của phương trình  $f(x) = 0$ .

Chẳng hạn ta tìm nghiệm gần đúng của phương trình  $x^4 - x - 78 = 0$  trong đoạn  $[2, 5]$ , ta có một số cách biến đổi thuận tiện cho phép lặp như sau:

**Cách 1.** Chúng ta đưa về phương trình  $x = x^4 - 78$  ở đây  $g(x) = x^4 - 78$  ta có  $|g'(x)| = 4|x^3|$  nhưng nó không thỏa mãn bị chặn bởi một giá trị  $q < 1$  với  $\forall x \in [2, 5]$  nên nếu ta dùng cách lặp này để tính thì không hội tụ về nghiệm.

**Cách 2.** Chúng ta đưa về phương trình  $x = \sqrt[4]{x+78}$  ở đây  $g(x) = \sqrt[4]{x+78}$  ta có  $|g'(x)| = \frac{1}{4(x+78)^{3/4}}$  luôn nhỏ hơn 1 với  $\forall x \in [2, 5]$  nên ta dùng cách này để tìm nghiệm gần đúng với sai số  $\varepsilon$  cho trước. Chúng ta xây dựng thuật toán lặp tới khi khoảng cách giữa hai giá trị liên tiếp nhỏ hơn sai số  $\varepsilon$  thì dừng.

**Chương trình:** Tìm nghiệm của phương trình  $x^4 - x - 78 = 0$  in ra nghiệm gần đúng là 3.000170 trong khi nghiệm đúng là 3.0.



```
#include<stdio.h>
#include<math.h>
double g(double x){return pow(x+78,0.25);}
int main()
{
    double u=5,v,epsilon=0.001;
    while(1)
    {
        v=g(u);
        if(fabs(u-v)<=epsilon) break;
        u=v;
    }
    printf("Nghiem gan dung %lf",u);
}
```

### 1.1.4. Tính đa dạng

Mỗi bài toán có nhiều cách giải hay thuật toán khác nhau, chúng ta luôn mong muốn tìm được thuật toán tốt nhất có thể cho bài toán đó.

**Ví dụ:** Bài toán cổ vũ

Trong một cuộc thi tài có 2 đối thủ mặc đai xanh và đai đỏ thi đấu với nhau, các cổ động viên đứng xếp thành một hàng dọc nếu cổ vũ cho đối thủ nào thì tay cầm cờ màu đai của đối thủ đó. Bạn hãy tìm một đoạn dài nhất liên tục các cổ động viên sao cho số cổ vũ cho đối thủ đai xanh bằng số cổ vũ cho đối thủ đai đỏ. Bài toán đặt ra, bạn nhập vào một xâu ký tự gồm những ký tự X biểu thị cho màu Xanh và D biểu thị cho màu Đỏ chỉ ra độ dài đoạn dài nhất có số ký tự X bằng số ký tự D.

#### **Cách 1. thuật toán trực tiếp**

Giả sử xâu nhập vào là  $s[1]...s[n]$ , ban đầu giá trị  $max = 0$ , ta sẽ duyệt mọi đoạn con của xâu từ vị trí  $i$  tới vị trí  $j$  và kiểm tra xem đoạn đó có số ký tự 'X' bằng số ký tự 'D' không? nếu bằng nhau thì ta so sánh với  $max$  để tìm ra độ dài đoạn dài nhất. Với cách làm này sẽ có độ phức tạp  $O(n^3)$  do hai vòng lặp duyệt hai nút và một vòng lặp kiểm tra xem số ký tự xanh bằng số ký tự đỏ. Chúng ta có thể cải tiến việc kiểm tra bằng cách sử dụng thêm hai mảng

$x[1...n]$  đếm số ký tự 'X' và  $d[1...n]$  đếm số ký tự 'D' từ vị trí là từ 1 tới vị trí đang xét, suy ra số ký tự 'X' từ vị trí  $i$  đến vị trí  $j$  là  $x[j] - x[i - 1]$ , còn số ký tự 'D' từ vị trí  $i$  đến vị trí  $j$  là  $d[j] - d[i - 1]$  nếu bằng nhau thì đoạn sẽ thỏa mãn.

### **Chương trình**

```
#include<stdio.h>
#include<string.h>
int main()
{
    char s[100006];
    scanf("%s",s+1); // x[1...n] nen ta doc vao tu x+1
    long max=0,n=strlen(s+1),x[100006],d[100006];
    x[0]=d[0]=0;
    for(long i=1; i<=n; i++)
        if(s[i]=='X') {
            x[i]=x[i-1]+1;
            d[i]=d[i-1];
        }
        else {
            x[i]=x[i-1];
            d[i]=d[i-1]+1;
        }
    for(long i=1; i<n; i++)
        for(long j=i+1; j<=n; j++)
            if(x[j]-x[i-1]==d[j]-d[i-1])
                if(max<j-(i-1)) max=j-(i-1);
    printf("%ld",max);
}
```

Với cách này độ phức tạp về thời gian giảm xuống  $O(n^2)$  nhưng về không gian sẽ là  $O(n)$ . Vì số lượng ký tự 'D' có thể suy ra từ tổng số trừ đi số lượng ký tự 'X' nên bạn có thể giảm một mảng  $d[1...n]$  mà chỉ tính theo mảng  $x[1...n]$ .

**Cách 2. thuật toán dùng bảng băm**

Ta dùng một mảng số nguyên biểu diễn xâu đầu vào với giá trị 1 biểu thị ký tự ‘X’ và  $-1$  biểu thị ký tự ‘D’. Bài toán bây giờ là tìm độ dài một đoạn liên tục dài nhất có tổng bằng 0. Ta duyệt từ trái sang phải cộng dồn giá trị nếu một giá trị  $s$  cùng xuất hiện ở vị trí  $i$  và  $j$  chứng tỏ đoạn từ  $i+1$  đến  $j$  có tổng bằng 0.

Chẳng hạn với xâu đầu vào là “XXDXDXX” ta có tổng biểu diễn như sau:

Vị trí	0	1	2	3	4	5	6	7
Giá trị xâu s		X	X	D	X	D	X	X
Giá trị số	0	1	1	$-1$	1	$-1$	1	1
Cộng dồn	0	1	2	1	2	1	2	3

Như vậy tổng cộng dồn tại vị trí 1, 3 bằng nhau chứng tỏ từ xâu từ vị trí 2 tới vị trí 3 cân bằng số ký tự ‘X’ và ‘D’. Tương tự như vậy vị trí 3, 5 hoặc 1, 5 hoặc 2, 4 hoặc 4, 6 hoặc 2, 6 bằng nhau thì suy ra được đoạn dài nhất cân bằng ký tự ‘X’ và ‘D’ là từ 2 đến 5 hoặc 3 đến 6.

Để lập trình ta dùng mảng  $M$  để lưu tổng bằng bao nhiêu là từ đầu đến vị trí nào đó, ví dụ  $M[3] = 7$ ,  $M[1] = 1$ , tất nhiên  $M[1]$  cũng có thể bằng 3 hoặc 5 nhưng để tìm độ dài nhất ta chỉ lưu vị trí đầu tiên. Chính vì việc băm theo tổng để lưu vị trí ta gọi cách này là sử dụng bảng băm. Chú ý rằng nếu xâu bắt đầu bằng một loạt ký tự ‘D’ có thể dẫn tới tổng âm nên mảng có chỉ số âm, khắc phục điều này ta cho con trỏ trỏ vào giữa một mảng cho trước có bộ nhớ cấp phát gấp đôi để lấy được chỉ số âm. Hơn nữa mảng lưu các số 1 và  $-1$  chỉ là để phân tích ý tưởng khi lập trình ta cộng dồn luôn thì không cần đến mảng này.

### Chương trình

```
#include<stdio.h>
#include<string.h>
char s[100006];
long M1[200006],*M=M1+100000; // M trỏ vào giữa mảng
int main()
{
    scanf("%s",s+1);
    long t=0,m=0,n=strlen(s+1);
    for(long i=0; i<=200000; i++) M1[i]=-1;
    M[0]=0;
    for(long i=1; i<=n; i++)
    {
        t+=s[i]=='X'?1:-1;
        if(M[t]<0) M[t]=i;
        else if(m<i-M[t]) m=i-M[t];
    }
    printf("%ld",m);
}
```

Với cách làm này chúng ta duyệt xâu chỉ một lần nên độ phức tạp  $O(n)$  và dùng một mảng phụ  $O(2n)$  với  $n$  là độ dài xâu. Trong lập trình nếu không thích tự quản lý mảng băm, ta có thể dùng cấu trúc dữ liệu **unordered\_map** khi đó chương trình được lập trình như sau:

### Chương trình

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    char s[1000006];
    unordered_map<long,long> M;
    scanf("%s",s);
    long t=0,m=0,n=strlen(s);
    M[0]=-1;
```

```
for(long i=0; i<n; i++)
{
    t+=x[i]=='X'?1:-1;
    if(M.find(t)==M.end()) M[t]=i;
    else
    {
        long k=M[t];
        if(m<i-k) m=i-k;
    }
}
printf("%ld",m);
}
```

### ***1.1.5. Tính phổ dụng***

Một tư tưởng thuật toán hóa có thể áp dụng giải nhiều bài toán trong một lớp các bài toán nào đó.

#### **Ví dụ 1:** Thuật toán tìm kiếm nhị phân

Bài toán nhập vào dãy  $a_1, a_2, \dots, a_n$  đã sắp xếp không giảm và một giá trị  $x$  chỉ ra 1 vị trí của  $x$  nếu nó xuất hiện trong dãy.

Tổng quát hóa ý tưởng thuật toán là tìm  $x$  trong dãy  $a$  từ vị trí  $L$  đến vị trí  $R$ , ban đầu  $L = 1, R = n$ , ta cắt đôi đoạn  $[L, R]$  bởi điểm giữa  $M = (L + R)/2$  (phép chia nguyên), nếu  $x = a_M$  thì ta chỉ ra vị trí là  $M$ , nếu  $x < a_M$  vì dãy đã sắp xếp tăng dần nên nếu có  $x$  sẽ nằm trong đoạn  $[L, M-1]$  ta gán lại  $R = M-1$ , ngược lại  $x > a_M$  thì gán lại  $L = M+1$  để thu nhỏ lại không gian tìm kiếm và lặp lại bước lặp tới khi tìm thấy  $x$  hoặc tới khi không gian tìm kiếm rỗng ( $L > R$ ) thì dừng.

***Hàm tìm kiếm trả về 0 nếu không tìm thấy  $x$***

```
#include<bits/stdc++.h>
#define FOR(i,a,b) for(int i=a;i<=b;i++)
using namespace std;
int TimKiemNhiPhan(int n, int*a, int x){
    int L =1, R =n ;
    while(L<=R)
```

```
{
    int M = (L + R)/2;
    if(x==a[M]) return M;
    if(x < a[M] ) R= M - 1;
    else L = M + 1;
}
return 0;
}
int main()
{
    int A[100005],n,x;
    printf("Nhap vao so phan tu : ");
    scanf("%d",&n);
    FOR(i,1,n) printf("A[%d]",i),scanf("%d",A+i);
    sort(A+1,A+n+1); //sap xep tang dan
    printf("\nDay sau khi sap xep\n");
    FOR(i,1,n) printf("\nA[%d] = %d",i,A[i]);
    printf("\nNhap gia tri x muon tim : ");
    scanf("%d",&x);
    int k=TimKiemNhiPhan(n,A,x);
    if(k==0) printf("Khong tim thay %d trong day",x);
    else printf("%d o vi tri A[%d]",x,k);
}
```

**Ví dụ 2:** Thuật toán chia đôi tìm nghiệm phương trình trong đoạn  $[a, b]$ .

Bài toán cho hàm số  $f(x) \in C[a, b]$  liên tục trên đoạn  $[a, b]$  và thỏa mãn  $f(a) \times f(b) \leq 0$ , theo tính chất của hàm liên tục thì tồn tại ít nhất một nghiệm  $x^* \in [a, b]$  tức là  $f(x^*) = 0$ , chúng ta tìm nghiệm xấp xỉ giá trị  $x^*$  với giá trị sai số  $\varepsilon$  cho trước.

Ta thực hiện ý tưởng cắt đôi đoạn  $[a, b]$  bởi điểm chia  $c = (b + a)/2$ , kiểm tra xem nếu  $f(a) \times f(c) \leq 0$  thì nghiệm thuộc đoạn  $[a, c]$  ta gán  $b = c$  hoặc ngược lại  $f(c) \times f(b) \leq 0$  thì nghiệm thuộc đoạn  $[c, b]$  ta gán  $a = c$  để thu gọn không gian tìm kiếm nghiệm. Sau đó, ta tiếp tục lặp lại công việc trên với đoạn  $[a, b]$  mới tới khi khoảng cách giữa  $a$  và  $b$  nhỏ hơn sai số  $\varepsilon$  cho trước thì dừng lại và thu được nghiệm gần đúng với sai số cho trước.

Minh họa giải nghiệm gần đúng phương trình  $\sin x = 0$  với sai số  $\varepsilon = 0.0001$  trên đoạn  $[3.0, 3.5]$  để tìm ra số  $\pi$  như sau.

```
#include<iostream>
#include<cmath>
using namespace std;
typedef double Ham(double);
double ChiaDoi(double a, double b, Ham f, double ep =1e-4)
{
    while (b - a > ep )
    {
        double c = (b + a)/2;
        if( f(a)* f(c) <= 0 ) b= c;
        else a =c;
    }
    return (b +a)/2;
}
int main()
{
    cout<<"pi = " << ChiaDoi(3,3.5,sin,0.0001);
}
```

Kết quả chạy chương trình ta thu được  $\pi = 3.14157$ .

#### 1.1.6. Tính hình thức hóa

Thuật toán chỉ dễ sử dụng được nếu nó biểu diễn được một cách hình thức bằng ngôn ngữ lập trình nào đó.

**Ví dụ:** Phương pháp hình thang tính gần đúng tích phân  $I = \int_a^b f(x)dx$ .

Ta chia đoạn  $[a, b]$  thành  $n$  đoạn con bằng nhau có độ dài là  $h = (b-a)/n$  bởi các điểm chia  $x_i = a + i \times h$  với  $\forall i = 0, \dots, n$ . Trên mỗi đoạn  $[x_{i-1}, x_i]$  ta xấp xỉ tích phân bằng diện tích hình thang vuông giới nội bởi các đường

$x = x_{i-1}, x = x_i, y = 0$  và  $y = f(x)$  tức là  $\int_{x_{i-1}}^{x_i} f(x)dx \approx \frac{f(x_{i-1}) + f(x_i)}{2} h$  suy ra

$$\begin{aligned}\int_a^b f(x)dx &= \sum_{i=1}^n \int_{x_{i-1}}^{x_i} f(x)dx \approx \sum_{i=1}^n \frac{f(x_{i-1}) + f(x_i)}{2} h \\&= \frac{h}{2} ((f(x_0) + f(x_1)) + \dots + (f(x_{n-1}) + f(x_n))) \\&= \frac{h}{2} \left( f(a) + \sum_{i=1}^{n-1} 2f(x_i) + f(b) \right) = h \left( \frac{f(a) + f(b)}{2} + \sum_{i=1}^{n-1} f(a + ih) \right)\end{aligned}$$

Chương trình sau minh họa tính

$$I = \int_0^1 \frac{4dx}{x^2 + 1} = 4 \arctan x \Big|_0^1 = 4 \left( \frac{\pi}{4} - 0 \right) = \pi \quad \text{với số điểm chia } n = 1000000 \text{ ra kết}$$

quả là “Giá trị tích phân  $I = 3.14159$ ”.

```
#include<iostream>
using namespace std;
double HinhThang(double a,double b,double f(double),int n)
{
    double h =(b - a)/n,S = 0;
    for (int i =1; i < n; i++)
        S += f(a +i * h);
    return (S + f(a)/2 + f(b)/2) * h;
}
double f(double x) {return 4/(1+ x*x );}
int main()
{
    cout<<"Tich phan I="<<HinhThang(0,1,f,1000000);
}
```

## 1.2. Biểu diễn thuật toán

### 1.2.1. Biểu diễn đặc tả tự nhiên

Sử dụng ngôn ngữ tự nhiên liệt kê từng bước thuật toán

**Ví dụ:** Thuật trừ tìm ước chung lớn nhất của hai số nguyên dương a, b

Chúng ta áp dụng bổ đề: Với hai số nguyên dương a, b với  $a > b$  thì ước chung lớn nhất của a và b bằng ước chung lớn nhất của  $a - b$  và b. Do đó thuật toán ta biến đổi số lớn thành hiệu 2 số đến khi bằng nhau thì ra ước chung lớn nhất.



### Thuật toán

**Bước 1.** Nhập vào hai số nguyên dương  $a, b$

**Bước 2.** Kiểm tra nếu  $a \neq b$  thì:

Nếu  $a > b$  ta đặt  $a = a - b$  và quay lại Bước 2

Ngược lại ta đặt  $b = b - a$  và quay lại Bước 2

**Bước 3.** Xuất ra giá trị  $a$  hoặc  $b$ .



### Chương trình



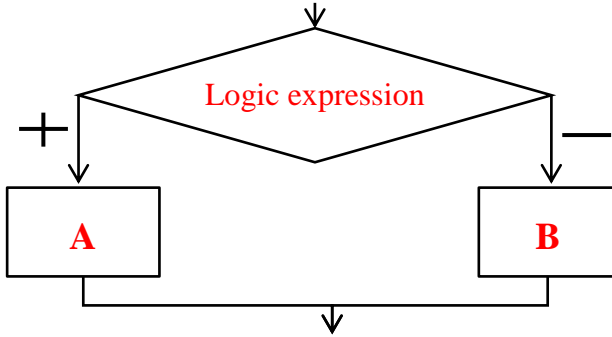
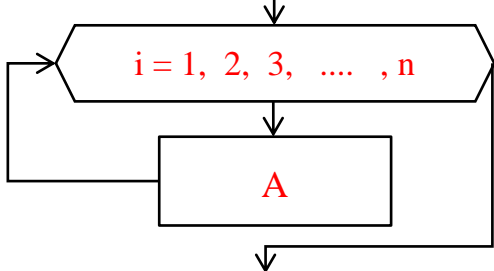
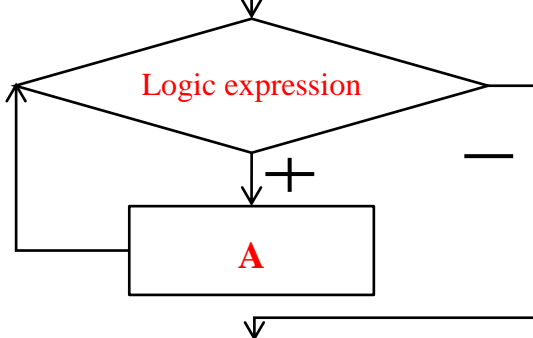
```
#include<stdio.h>
#define input(n) printf("Nhap so %s : ",#n);
scanf("%u",&n);
int main()
{
    unsigned int a, b;
    Buoc1:
        input(a);
        input(b);
    Buoc2:
        if(a!=b)
        {
            if(a>b) {a=a-b; goto Buoc2; }
            else {b=b-a; goto Buoc2;}
        }
    Buoc3:    printf("Uoc chung lon nhat la %d",a);
}
```

Phương pháp này dễ mô tả nhưng các bước nhiều khi không rõ ràng và mang tính cảm tính.

#### 1.2.2. Biểu diễn bằng sơ đồ khối

Sử dụng các ký hiệu, các sơ đồ trực quan miêu tả thuật toán

STT	Mẫu hình	Ý nghĩa
1		Bắt đầu và kết thúc thuật toán
2		Nhập và xuất dữ liệu

3		Hướng đi của dữ liệu
4		Thực hiện tính toán hoặc gán dữ liệu
5		Rẽ nhánh nếu biểu thức logic đúng thì thực hiện A ngược lại sai thì thực hiện B
6		Vòng lặp xác định thực hiện lần lượt n lần thao tác A
7		Vòng lặp không xác định nếu biểu thức logic đúng thì thực hiện A sau đó quay lại lặp tới khi biểu thức logic sai thì ra khỏi vòng lặp

**Ví dụ:** Nhập số nguyên dương n và tính giá trị biểu thức

$$S = \begin{cases} 1 \times 4 \times 7 \times \dots \times n & \text{khi } n \equiv 1 \pmod{3} \\ 2 \times 5 \times 8 \times \dots \times n & \text{khi } n \equiv 2 \pmod{3} \\ 3 \times 6 \times 9 \times \dots \times n & \text{khi } n \equiv 0 \pmod{3} \end{cases}$$

Thuật toán tính giá trị của S ta thay vì nhân xuôi ta sẽ xuất phát  $S = 1$  và nhân ngược từ n rồi  $n - 3$  đến  $n - 6 \dots$  cứ như vậy chừng nào n còn không âm thì còn nhân vào S theo sơ đồ và chương trình sau:

Sơ đồ	Chương trình
<pre> graph TD     Begin([Begin]) --&gt; Input[/Input n ∈ ℤ⁺/]     Input --&gt; S1[S = 1]     S1 --&gt; Decision{n &gt; 0}     Decision -- "+" --&gt; Process[S = S * n n = n - 3]     Process --&gt; Decision     Decision -- "-" --&gt; Output[/Output S ∈ ℤ⁺/]     Output --&gt; End([End])         </pre>	<pre> #include&lt;stdio.h&gt; int main() {     int n,s=1;     printf("Nhap vao n = ");     scanf("%d",&amp;n);     while(n&gt;0) {s*=n; n-=3;}     printf("S = %d",s); }         </pre>

Biểu diễn thuật toán bằng sơ đồ khối trực quan, dễ chuyển sang chương trình nhưng khá cồng kềnh khi mô tả bài toán phức tạp.

### 1.2.3. Biểu diễn bằng tựa ngôn ngữ lập trình Pascal

Ngôn ngữ lập trình Pascal là ngôn ngữ lập trình hướng chức năng nên rất dễ dùng miêu tả các quy trình, thủ tục hay thuật toán.

**Ví dụ:** Nhập vào dãy số nguyên  $a_1, a_2, \dots, a_n$  hãy tìm khoảng cách lớn nhất giữa hai số trong dãy với điều kiện số lớn đứng trước số bé, trong trường hợp không có in ra màn hình -1.

**Thuật toán trực tiếp:** Chúng ta tìm giá trị lớn nhất của của hiệu tất cả các cặp số trước trừ số sau tức là tìm  $\max_{1 \leq i < j \leq n} (a_i - a_j)$  nếu giá trị này không âm thì xuất ra -1 bằng cách duyệt hai vòng lặp theo giả mã sau:

```
Function Max1(A: array[1..100] of integer; n: integer): integer;  
Begin  
    Var i, j, M: integer;  
  
    M := -1;  
    For i := 1 to n do  
        For j := i+1 to n do  
            If M < A[i] - A[j] then M := A[i] - A[j];  
        If M <= 0 then M := -1;  
    Max1 := M;  
End;
```

### **Chương trình**

```
#include<stdio.h>  
#define FOR(i,a,b) for(int i=a;i<=b;i++)  
int main()  
{  
    int A[100005], n, M=-1;  
    printf("Nhap vao so phan tu : ");  
    scanf("%d", &n);  
    FOR(i, 1, n) printf("A[%d]", i), scanf("%d", &A[i]);  
    FOR(i, 1, n)  
        FOR(j, i+1, n)  
            if(M < A[i] - A[j]) M = A[i] - A[j];  
    printf("%d", (M <= 0 ? -1 : M));  
}
```

**Thuật toán cải tiến:** Để tìm giá trị lớn nhất của các hiệu số trước trừ sau, với mỗi một chỉ số  $j$  ( $2 \leq j \leq n$ ) chúng ta tìm giá trị lớn nhất của các số đứng trước  $j$  để trừ đi  $a_j$ , với  $j = 2$  chúng ta khởi gán  $\text{Max} = a_1$ , sau đó ở bước  $j - 1$  ta đã tìm được  $\text{Max} = \text{Max}(a_1, a_2, \dots, a_{j-1})$  nên sang bước  $j$  sau khi tính xong so sánh  $\text{Max}$  này với  $a_j$  để cập nhật  $\text{Max} = \text{Max}(a_1, a_2, \dots, a_j)$  rồi tính ở bước này theo giả mã:

```
Function Max2(A: array[1..1000] of integer; n: integer): integer;  
Begin  
    Var i, j, M, Max: integer;  
    M := -1;  
    Max := A[1];  
    For j := 2 to n do  
        Begin  
            If M < Max - A[j] then M := Max - A[j];  
            If Max < A[j] then Max := A[j];  
        End;  
    If M <= 0 then M := -1;  
    Max2 := M;  
End;
```

### ***Chương trình***

```
#include<stdio.h>  
#define FOR(i,a,b) for(int i=a;i<=b;i++)  
int main()  
{  
    int A[100005], n, M=-1, Max;  
    printf("Nhap vao so phan tu : ");  
    scanf("%d", &n);  
    FOR(i, 1, n) printf("A[%d]=", i), scanf("%d", &A[i]);  
    Max = A[1];  
    FOR(j, 2, n){  
        if(M < Max - A[j]) M = Max - A[j];  
        if(Max < A[j]) Max = A[j];  
    }  
    printf("%d", (M <= 0 ? -1 : M));  
}
```

Bằng cảm giác thuật toán thứ hai tốt hơn thuật toán thứ nhất để đánh giá được độ phức tạp tiếp tục chúng ta nghiên cứu mục đánh giá thuật toán.

### **1.3. Độ phức tạp thuật toán**

#### **1.3.1. Khái niệm**

Độ phức tạp của 1 thuật toán là nguồn tài nguyên mà thuật toán đó sử dụng gồm tài nguyên về không gian và tài nguyên về thời gian.

+ Tài nguyên về không gian: số bit hoặc số byte chứa các thông tin dữ liệu đầu vào, đầu ra; các bộ nhớ trung gian như các biến, mảng mà thuật toán đó sử dụng.

+ Tài nguyên về thời gian: số phép toán cơ bản mà thuật toán tính toán gồm phép toán bit, phép toán logic, phép toán số học, phép so sánh, phép gán, phép tăng, phép giảm, phép tham chiếu hoặc cấp phát và giải phóng bộ nhớ ...

**Ví dụ:** Cho số nguyên dương  $n$  tính số Fibonacci thứ  $n$  theo công thức:

$$F_n = \begin{cases} 1 & \text{khi } n = 1, 2 \\ F_{n-1} + F_{n-2} & \text{khi } n \geq 3 \end{cases}$$

#### **Chương trình**

```
#include<stdio.h>
long Fibonacci(int n)
{
    long *F= new long[n+1];
    F[1] = F[2] = 1;
    for(int k=3;k<=n;k++) F[k] = F[k-1] + F[k-2];
    long x= F[n];
    delete F;
    return x;
}
int main()
{
    int n;
    printf("Nhập vào n = ");scanf("%d",&n);
    printf("\nF[%d] = %lld",n,Fibonacci(n));
}
```

**Độ phức tạp về không gian:**

$$K(n) = \text{sizeof}(n) + \text{sizeof}(F) + \text{sizeof}(k) + \text{sizeof}(x);$$

$$= 4 + (n+1)*4 + 4 + 4 = 4n + 16 = O(n).$$

**Độ phức tạp về thời gian**

Hàm	Phép toán cơ bản	Tổng
<code>long Fibonacci(int n)</code> <code>{</code>		
<code>long *F= new long[n+1];</code>	long =, new, +	4
<code>F[1] = F[2] = 1;</code>	[], = , [], =	4
<code>int k=3;</code>	int, =	2
<code>for(; k &lt;= n; k++)</code> <code>F[k] = F[k-1] + F[k-2];</code>	<=, ++, [], =, [], -, +, [], -	(n-2)*9
<code>long x= F[n];</code>	long, =, []	3
<code>delete F;</code>	Delete	1
<code>return x;</code> <code>}</code>		1

$$\text{Tổng số } T(n) = 4 + 4 + 2 + 9(n - 2) + 3 + 1 + 1 = 9n - 3 = O(n)$$

Hiện nay, với sự phát triển của công nghệ sản xuất phần cứng, không gian lưu trữ được cải thiện rất lớn tới mức mà hầu như khi đánh giá độ phức tạp thuật toán người ta ít quan tâm đến độ phức tạp không gian, mà chỉ quan tâm tới độ phức tạp về thời gian.

**1.3.2. Đánh giá độ phức tạp thời gian**

Để đánh giá độ phức tạp về thời gian, thường có hai cách đánh giá:

**Theo xu hướng lý thuyết:** đếm số phép toán cơ bản phải thực hiện, điều này phụ thuộc vào cách cài đặt chi tiết của thuật toán và phụ thuộc vào dữ liệu đầu vào. Chẳng hạn khi chúng ta kiểm tra một số có phải là số nguyên tố không? Nếu đầu vào là một số chẵn ta kết luận được luôn? Nhưng nếu là số lẻ và hơn nữa lại đúng là số nguyên tố thì tốn rất nhiều thời gian để kiểm tra. Như vậy thông thường khi đánh giá chúng ta thường xem xét độ phức tạp tốt nhất, xấu nhất và trung bình tùy theo dữ liệu đầu vào.

**Theo xu hướng thực nghiệm:** viết chương trình hoàn chỉnh và cho chạy rồi đo thời gian. Với cách làm này thì lại phụ thuộc vào cấu hình máy

tính, muốn so sánh độ phức tạp ta phải chạy trên cùng một máy tính và cùng dữ liệu đầu vào giống nhau.

**Ví dụ:** Thuật toán kiểm tra số nguyên dương  $n$  có là số nguyên tố không?

Thuật toán ở đây ta chia ra các trường hợp

Nếu  $n < 2$  thì không nguyên tố

Ngược lại nếu  $n = 2$  thì nguyên tố

Ngược lại nếu  $n$  chia hết cho 2 thì không nguyên tố

Ngược lại  $n$  lẻ lớn hơn 2 nếu  $n$  phân tích thành  $n = a \times b$  trong đó  $a, b \in \mathbb{Z}$  và  $a \leq b$  suy ra  $a \leq \sqrt{n}$  vì nếu ngược lại  $b \geq a \geq \sqrt{n}$  thì hiển nhiên  $a \times b > n$  sinh ra mâu thuẫn. Nên để kiểm tra  $n$  lẻ có là nguyên tố không ta sẽ kiểm tra xem  $n$  có chia hết cho số lẻ nào không vượt quá  $\sqrt{n}$  nếu có chia hết bất cứ số nào thì  $n$  không nguyên tố ngược lại  $n$  là số nguyên tố.

### Chương trình

```
#include<stdio.h>
typedef long long LL;
bool KTNT(LL n)
{
    if(n==2) return true;
    if(n<2 || n%2 == 0) return false;
    for(LL i=3; i*i<=n; i+=2) if(n%i==0) return false;
    return true;
}
int main()
{
    LL n=1000000007;
    printf(KTNT(n)?"%lld nguyên to":"%lld không nguyên to",n);
}
```

Theo xu hướng lý thuyết: trường hợp tốt nhất là  $O(1)$ ; trường hợp xấu nhất  $O(\sqrt{n} / 2)$ .



Theo xu hướng thực nghiệm khi kiểm tra 1000000007 đúng là số nguyên tố chạy trên máy Core i7, có RAM 2Gb thì thời gian khoảng 0.0005 giây.

### **1.3.3. Các kí hiệu tiệm cận: $O, \Omega, \Theta$**

Trong phân tích, thiết kế thuật toán chúng ta ít quan tâm tới độ phức tạp khi kích thước dữ liệu đầu vào nhỏ mà quan tâm nhiều hơn khi kích thước dữ liệu đầu vào lớn. Chính vì lẽ đó khi kích thước rất lớn chúng ta đưa vào các ký hiệu tiệm cận biểu thị độ phức tạp của thuật toán.

#### **a. Ký hiệu tiệm cận trên $O$**

Giả sử  $f: \mathbb{R}^+ \rightarrow \mathbb{R}^+$  và  $g: \mathbb{R}^+ \rightarrow \mathbb{R}^+$  là hai hàm xác định dương ta nói  $g(n)$  có tiệm cận trên là  $O(f(n))$  ký hiệu  $g(n) \in O(f(n))$  hoặc  $g(n) = O(f(n))$  nếu tồn tại hằng số thực dương  $c \in \mathbb{R}^+$  và tồn tại số thực dương  $n_0$  ta có  $g(n) \leq c \times f(n)$  với mọi  $n \geq n_0$ . Ở đây  $O(f(n))$  là một tập các hàm có tiệm cận trên là  $f(n)$ .

**Khái niệm:** Giả sử  $f: \mathbb{R}^+ \rightarrow \mathbb{R}^+$  là một hàm xác định dương ta ký hiệu tập các hàm có tiệm cận trên  $f(n)$  là  $O(f(n))$ :

$$O(f(n)) = \{ g: \mathbb{R}^+ \rightarrow \mathbb{R}^+ \mid \exists n_0 \text{ đủ lớn, } \exists c \in \mathbb{R}^+ \text{ đều có } g(n) \leq cf(n) \quad \forall n \geq n_0 \}$$

**Ví dụ 1:** Ta có  $3n^2 + 2n + 5 = O(n^3)$ , vì tồn tại  $n_0 = 1$ ,  $c = 10$  thì  $3n^2 + 2n + 5 \leq 10n^3$  với mọi  $n \geq n_0 = 1$ .

**Ví dụ 2.** Ta có  $3n^3 + 2n^2\sqrt{n} + 3 = O(n^3)$ , vì tồn tại  $n_0 = 1$ ,  $c = 8$  thì  $3n^3 + 2n^2\sqrt{n} + 3 \leq 8n^3$  với mọi  $n \geq n_0 = 1$ .

#### **b. Ký hiệu tiệm cận dưới $\Omega$**

Giả sử  $f: \mathbb{R}^+ \rightarrow \mathbb{R}^+$  và  $g: \mathbb{R}^+ \rightarrow \mathbb{R}^+$  là hai hàm xác định dương ta nói  $g(n)$  có tiệm cận dưới là  $\Omega(f(n))$  ký hiệu  $g(n) \in \Omega(f(n))$  hoặc  $g(n) = \Omega(f(n))$  nếu tồn tại hằng số thực dương  $c \in \mathbb{R}^+$  và tồn tại số thực

đương  $n_0$  ta có  $0 < c \times f(n) \leq g(n)$  với mọi  $n \geq n_0$ . Ở đây  $\Omega(f(n))$  là một tập các hàm có tiệm cận dưới là  $f(n)$ .

**Khái niệm:** Giả sử  $f: \mathbb{R}^+ \rightarrow \mathbb{R}^+$  là một hàm xác định dương ta ký hiệu tập các hàm có tiệm cận dưới  $f(n)$  là  $\Omega(f(n))$ :

$$\Omega(f(n)) = \{ g: \mathbb{R}^+ \rightarrow \mathbb{R}^+ \mid \exists n_0 \text{ đủ lớn, } \exists c \in \mathbb{R}^+ \text{ đều có } cf(n) \leq g(n) \forall n \geq n_0 \}$$

**Ví dụ 1:** Ta có  $\frac{1}{3}n^3 + 2\sqrt{n} + 10 = \Omega(n^2)$ , vì tồn tại  $n_0 = 1$ ,  $c = 1/3$  thì

$$\frac{1}{3}n^2 \leq \frac{1}{3}n^3 + 2\sqrt{n} + 10 \text{ với mọi } n \geq n_0 = 1.$$

**Ví dụ 2.** Ta có  $\sqrt{n} = \Omega(\log_2 n)$ , vì tồn tại  $n_0 = 64$ ,  $c = 1$  thì  $\log_2 n \leq \sqrt{n}$  với mọi  $n \geq n_0 = 64$ .

### c. Ký hiệu tiệm cận chặt $\Theta$

Giả sử  $f: \mathbb{R}^+ \rightarrow \mathbb{R}^+$  và  $g: \mathbb{R}^+ \rightarrow \mathbb{R}^+$  là hai hàm xác định dương ta nói  $g(n)$  có tiệm cận dưới là  $\Theta(f(n))$  ký hiệu  $g(n) \in \Theta(f(n))$  hoặc  $g(n) = \Theta(f(n))$  nếu tồn tại 2 hằng số thực dương  $c_1, c_2 \in \mathbb{R}^+$  và tồn tại số thực dương  $n_0$  ta có  $0 < c_1 \times f(n) \leq g(n) \leq c_2 \times f(n)$  với mọi  $n \geq n_0$ . Ở đây  $\Theta(f(n))$  là một tập các hàm có tiệm cận chặt là  $f(n)$ .

**Khái niệm:** Giả sử  $f: \mathbb{R}^+ \rightarrow \mathbb{R}^+$  là một hàm xác định dương ta ký hiệu tập các hàm có tiệm cận chặt  $f(n)$  là  $\Theta(f(n))$ :

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

$$\{ g: \mathbb{R}^+ \rightarrow \mathbb{R}^+ \mid \exists n_0 \text{ đủ lớn, } \exists c \in \mathbb{R}^+ \text{ đều có } c_1 f(n) \leq g(n) \leq c_2 f(n) \forall n \geq n_0 \}$$

**Ví dụ 1:** Ta có  $\frac{n(n+1)}{2} = \Theta(n^2)$ , vì tồn tại  $n_0 = 1$ ,  $c_1 = 1/4$ ,  $c_2 = 1$  thì

$$\frac{1}{4}n^2 \leq \frac{n(n+1)}{2} \leq 1n^2 \text{ với mọi } n \geq n_0 = 1.$$

**Ví dụ 2.** Ta có  $\log_3 n = \Theta(\log_2 n)$ , vì tồn tại  $n_0 = 1$ ,  $c_1 = c_2 = \log_2 3$  thì  $\log_3 2 \times \log_2 n \leq \log_3 n \leq \log_3 2 \times \log_2 n$  với mọi  $n \geq n_0 = 1$ . Từ ví dụ này từ

nay về sau khi xét đến độ phức tạp hàm logarit ta sẽ không quan tâm đến cơ số.

**Tính chất:** Giả sử  $f, g, h$  là các hàm xác định dương và  $c$  là hằng số thực dương ta có:

- Phản xạ  $f(n) = O(c.f(n)); f(n) = \Omega(c.f(n)); f(n) = \Theta(c.f(n))$

- Chuyển đổi:  $f(n) = O(g(n))$  khi và chỉ khi  $g(n) = \Omega(f(n))$

- Đối xứng:  $f(n) = \Theta(g(n))$  khi và chỉ khi  $g(n) = \Theta(f(n))$

- Bắc cầu:

Nếu  $f(n) = O(g(n))$  và  $g(n) = O(h(n))$  thì  $f(n) = O(h(n))$

Nếu  $f(n) = \Omega(g(n))$  và  $g(n) = \Omega(h(n))$  thì  $f(n) = \Omega(h(n))$

Nếu  $f(n) = \Theta(g(n))$  và  $g(n) = \Theta(h(n))$  thì  $f(n) = \Theta(h(n))$

- Tổng thể:  $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

### 1.3.4. Đánh giá độ phức tạp của thuật toán

#### a. Đánh giá thuật toán lặp

**Ví dụ 1:** Nhập số nguyên dương  $n$  và tính giá trị biểu thức

$$S = \begin{cases} 1 \times 4 \times 7 \times \dots \times n & \text{khi } n \equiv 1 \pmod{3} \\ 2 \times 5 \times 8 \times \dots \times n & \text{khi } n \equiv 2 \pmod{3} \\ 3 \times 6 \times 9 \times \dots \times n & \text{khi } n \equiv 0 \pmod{3} \end{cases}$$

**Chương trình:**

```
#include<stdio.h>
int main()
{
    int n,s=1;
    printf("Nhap vao n = ");
    scanf("%d",&n);
    for(;n>0;n-=3) s*=n;
    printf("S = %d",s);
}
```

Trong đoạn chương trình trên ta thấy rằng vòng lặp chạy số bước tuyến tính theo  $n$  mà chính xác là mất  $n/3$  bước nên độ phức tạp là  $\Theta(n/3) = \Theta(n)$ .

**Ví dụ 2: (Chặt nhị phân)** Nhập số nguyên không âm  $n$  kiểm tra xem  $n$  có là số chính phương không. Thuật toán ở đây chúng ta tìm xem từ 0 đến  $n$  có số nào bình phương bằng  $n$  không bằng phương pháp chặt nhị phân.

### Chương trình

```
#include<stdio.h>
bool ktcp(long long n)
{
    long long L=0,R=n,M;
    while(L<=R)
    {
        M=(L+R)/2;
        if(M*M==n) return true;
        if(M*M<n) L=M+1;
        else R=M-1;
    }
    return false;
}
int main()
{
    long long n;
    printf("Nhap vao n = ");
    scanf("%lld",&n);
    printf(ktcp(n)?"%d chinh phuong\n":"%d khong chinh
phuong\n",n);
}
```

Trong đoạn chương trình trên, ban đầu khoảng tìm kiếm sẽ là số phần tử trong đoạn  $[L,R]$  chính là đoạn  $[0, n]$  có  $n + 1$  phần tử. Mỗi bước lặp khoảng lặp bước sau chỉ bằng nửa khoảng lặp bước trước, nên bước đi không tuyến tính như trong ví dụ 1. Như vậy, bước 1 tìm trên  $n + 1$  phần tử, bước 2 tìm trên  $(n + 1)/2$  ... tới bước  $k$  tìm trên  $(n + 1) / 2^k$  chỉ còn 1 phần tử thì  $k = \log_2(n + 1)$ . Do đó độ phức tạp của thuật toán là  $\Theta(\log n)$ .

**Chú ý:** Chúng ta đánh giá độ phức tạp của thuật toán lặp cần phải chú ý khoảng cách bước lặp hoặc độ đo của tiến trình lặp.

+ **Quy tắc cộng:** Nếu thuật toán A được biểu diễn bởi chương trình P gồm hai đoạn chương trình P1 và P2 rời nhau thì:

$$T_A(n) = T_P(n) = T_{P1}(n) + T_{P2}(n) = \text{Max}(T_{P1}(n), T_{P2}(n))$$

+ **Quy tắc nhân:** Nếu thuật toán A được biểu diễn bởi chương trình P gồm hai đoạn chương trình P1 và P2 lồng nhau thì:

$$T_A(n) = T_P(n) = T_{P1}(n) * T_{P2}(n)$$

**Ví dụ:** Tìm điểm yên ngựa trên ma trận

Điểm yên ngựa của ma trận A cỡ  $n \times m$  và điểm mà giá tại vị trí đó là nhỏ nhất theo hàng và lớn nhất theo cột. Để tìm điểm yên ngựa ta duyệt tất cả các vị trí u, v và kiểm tra xem tại đó ma trận có đạt giá trị nhỏ nhất theo hàng và lớn nhất theo cột không?

### Chương trình

```
#include<stdio.h>
#define FOR(i,n) for(int i=1;i<=n;i++)
typedef int matran[100][100];
void Nhap(int &n,matran &A){
    printf("Nhap vao co cua matran n : ");
    scanf("%d",&n);
    FOR(i,n)
        FOR(j,n){
            printf("A[%d][%d] = ",i,j);
            scanf("%d",&A[i][j]);
        }
}
bool MinHang(int n,matran &A,int u,int v){
    FOR(i,n) if(A[i][v]<A[u][v]) return false;
    return true;
}
bool MaxCot(int n,matran &A,int u,int v){
    FOR(j,n) if(A[u][j]>A[u][v]) return false;
```

```
        return true;
    }
    void YenNgua(int n,matran &A){
        FOR(u,n)
            FOR(v,n)
                if(MinHang(n,A,u,v) && MaxCot(n,A,u,v))
                    printf("\nDiem yen ngua tai A[%d][%d]",u,v);
    }
    int main()
    {
        int n;
        matran A;
        Nhap(n,A);
        YenNgua(n,A);
    }
```

Theo quy tắc cộng thì độ phức tạp thuật toán sẽ phụ thuộc và cỡ của ma trận nhập vào là  $n$ . Ta có  $T(n) = T_{Nhập}(n) + T_{YenNgua}(n)$ , hàm Nhap có hai vòng lặp lồng nhau nên  $T_{Nhập}(n) = \Theta(n^2)$  còn hàm YenNgua có hai vòng lặp lồng nhau và mỗi lần lặp lại lồng bên trong làm hàm MinHang có độ phức tạp  $T_{MinHang}(n) = \Theta(n)$ , tương tự hàm MaxCot có độ phức tạp  $T_{MaxCot}(n) = O(n)$ , do đó  $T_{YenNgua}(n) = \Theta(n * n * (n + n)) = \Theta(n^3)$  nên độ phức tạp chương trình sẽ là  $T(n) = \Theta(n^2) + \Theta(n^3) = \Theta(n^3)$ .

### ***b. Đánh giá thuật toán Đệ Quy***

Mỗi một thuật toán đệ quy đều dựa trên một thủ tục hoặc một biểu thức truy hồi tính toán, để đánh giá độ phức tạp của thuật toán đệ quy ta cũng xây dựng biểu thức truy hồi đánh giá sau đó dùng các kỹ thuật đánh giá biểu thức truy hồi để đánh giá độ phức tạp của thuật toán đệ quy đó.

**Ví dụ 1:** Tính giai thừa của số nguyên không âm  $n! = 1 \times 2 \times 3 \times \dots \times n$

$$\text{Trước hết ta có biểu thức truy hồi} \quad n! = \begin{cases} 1 & \text{khi } n = 0 \\ n \times (n-1)! & \text{khi } n > 0 \end{cases}$$

**Chương trình**

```
#include<stdio.h>
long GiaiThua(int n)
{
    if(n == 0) return 1;
    return n * GiaiThua(n-1);
}
int main()
{
    int n;
    printf("Nhap vao n = ");scanf("%d",&n);
    printf("\n%d! = %lld",n,GiaiThua(n));
}
```

Ta đặt  $T(n)$  là độ phức tạp của thuật toán khi tính  $n!$  ta có biểu thức đánh giá độ phức tạp:

$$T(n) = \begin{cases} c_0 & \text{khi } n = 0 \\ T(n-1) + c & \text{khi } n > 0 \end{cases}$$

Trong đó khi  $n = 0$  thì thuật toán phải thực hiện 2 phép toán là phép so sánh  $n == 0$  và phép trả về kết quả (return) nên  $c_0 = 2$ , còn khi  $n > 0$  thì thuật toán sẽ thực hiện phép so sánh  $n == 0$ , phép nhân, phép trừ  $(n-1)$  và phép trả về (return) và thời gian gọi đệ quy chạy  $T(n-1)$  nên ở đây  $c = 4$ .

Ta thực hiện phép thế dần biểu thức  $T(n) = T(n-1) + c$  ta có

$$\begin{aligned} T(n) &= T(n-1) + c = T(n-2) + c + c = T(n-2) + 2c = \dots = T(n-n) + nc = \\ T(0) + nc &= c_0 + nc = \Theta(n). \end{aligned}$$

Vậy độ phức tạp của thuật toán là  $\Theta(n)$ .

**Ví dụ 2:** Cho biểu thức truy hồi

$$F(x,n) = \begin{cases} 1 & \text{Khi } n = 0 \\ \sum_{i=1}^n \frac{x}{i} + F(2x+1, \lfloor n/2 \rfloor) \times F(-4x+1, \lfloor n/4 \rfloor) & \text{Khi } n > 0 \end{cases}$$

Hãy viết chương trình nhập vào số thực  $x$  và số nguyên không âm  $n$  tính giá trị của biểu thức.

**Chương trình đệ quy**

```

#include<stdio.h>
float F(float x, int n){
    if(n==0) return 1;
    float s=0;
    for(int i=1;i<=n;i++) s+=x/i;
    return s+F(2*x+1,n/2)*F(-4*x+1,n/4);
}
int main(){
    float x;
    int n;
    printf("Nhap vao so thuc x: ");scanf("%f",&x);
    printf("Nhap vao so nguyen khong am n:");
    scanf("%d",&n);
    printf("Gia tri bieu thuc F(x,n) = %f",F(x,n));
}

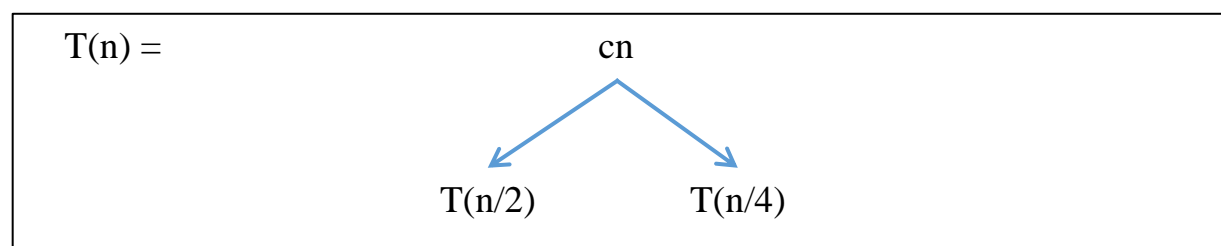
```

Đánh giá độ phức tạp về thời gian: Ta thấy rằng độ phức tạp sẽ phụ thuộc vào  $n$  ta đặt  $T(n)$  là độ phức tạp về thời gian của thuật toán, thời gian chạy sẽ mất một vòng lặp và hai lần gọi đệ quy một lần thời gian  $T(n/2)$  và một lần thời gian  $T(n/4)$  ta có:

$$T(n) = \begin{cases} c_0 & \text{Khi } n = 0 \\ \Theta(n) + T(\lfloor n/2 \rfloor) + T(\lfloor n/4 \rfloor) & \text{Khi } n > 0 \end{cases}$$

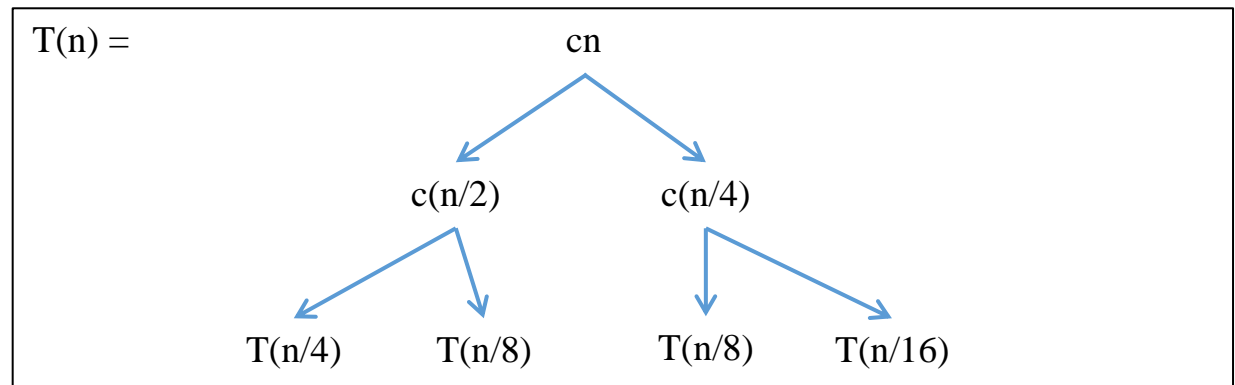
Để đơn giản ta xét  $T(n) = cn + T(n/2) + T(n/4)$  và thực hiện phép thế  $T(n/2) = cn/2 + T(n/4) + T(n/8)$  và  $T(n/4) = cn/4 + T(n/8) + T(n/16)$  cứ như vậy tiếp tục thế tiếp ta có các bước xây dựng cây đệ quy như sau:

**Bước 1:** Thay  $T(n) = cn + T(n/2) + T(n/4)$



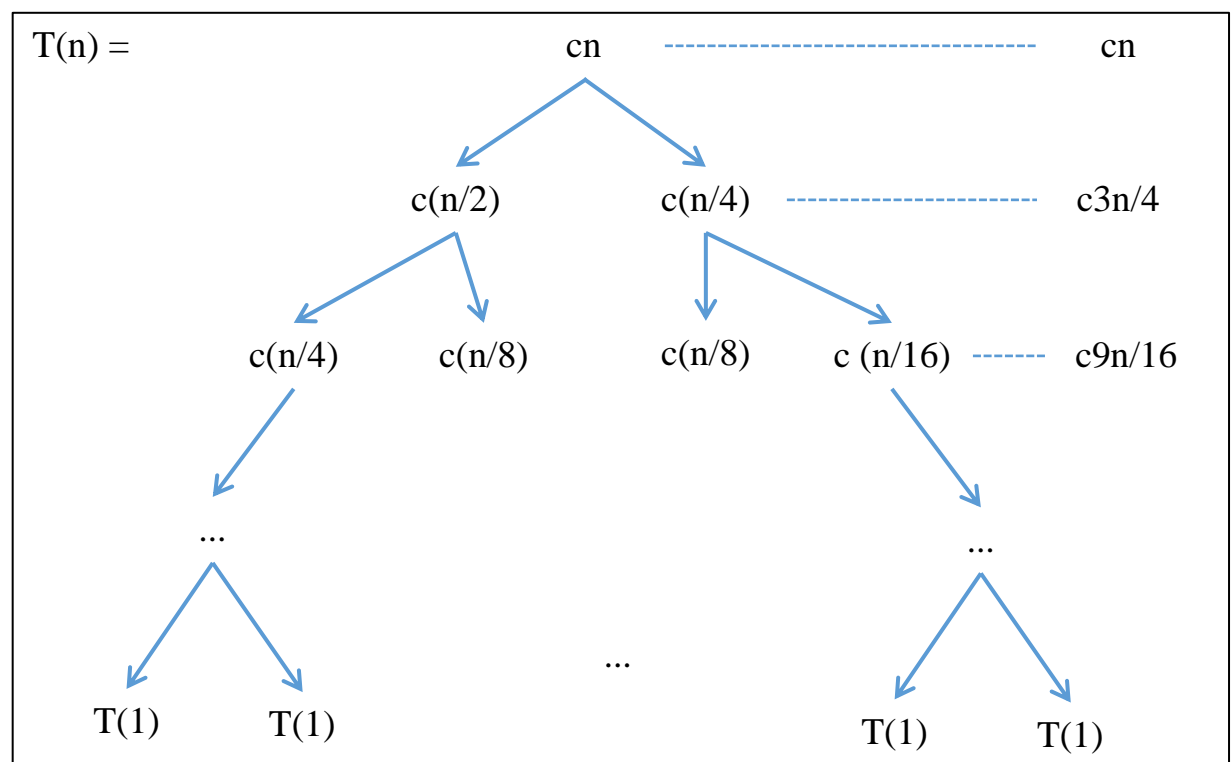


**Bước 2:** Tiếp tục thay  $T(n/2) = cn/2 + T(n/4) + T(n/8)$  và  $T(n/4) = cn/4 + T(n/8) + T(n/16)$ .



.....

Tiếp tục thế tiếp đến khi tất cả về  $T(1)$  thì dừng lại ta được cây đệ quy



Như vậy  $cn \leq T(n) \leq cn + c\frac{3}{4}n + c\frac{9}{16}n + \dots + c\left(\frac{3}{4}\right)^h n + \dots$  ta suy ra

$$cn \leq T(n) \leq cn \left( 1 + \frac{3}{4} + \frac{9}{16} + \dots + \left(\frac{3}{4}\right)^h + \dots \right) < cn \frac{1}{1 - 3/4} = 4cn$$

cho nên  $cn \leq T(n) \leq 4cn$  vậy theo định nghĩa thì  $T(n) = \Theta(n)$ .

Như vậy với những thuật toán đệ quy ta sẽ xây dựng biểu thức truy hồi để đánh giá sau đó chủ yếu sử dụng phép thế thông thường được biểu diễn bằng cây đệ quy để ước lượng đánh giá. Với phương pháp này độ tin cậy không cao phải vì cảm tính trông vào cây để ước lượng do đó nên sử dụng thêm phép thế để chứng minh toán học chặt chẽ.

Trong một số trường hợp nếu biểu thức truy hồi đánh giá có dạng  $T(n) = aT(n/b) + cn^k$  ta có thể dùng định lý thợ (Master theorem) thu gọn đánh biểu thức này.

**Định lý Master thu gọn (Master theorem):** Cho số nguyên  $n$  không âm và các hằng số thực  $a \geq 1, b > 1, k \geq 0$  và  $c > 0$  bất kỳ trong biểu thức truy hồi đánh giá

$$T(n) = \begin{cases} \Theta(1) & \text{Khi } n = 0, 1 \\ aT(n/b) + cn^k & \text{Khi } n > 1 \end{cases}$$

Ta có:

Nếu  $\log_b a > k$  hay  $a > b^k$  thì  $T(n) = \Theta(n^{\log_b a})$

Nếu  $\log_b a = k$  hay  $a = b^k$  thì  $T(n) = \Theta(n^k \log n)$

Nếu  $\log_b a < k$  hay  $a < b^k$  thì  $T(n) = \Theta(n^k)$

**Ví dụ 1:** Xác định độ phức tạp của biểu thức

$$T(n) = \begin{cases} \Theta(1) & \text{Khi } n = 0, 1 \\ 3T\left(\frac{n}{4}\right) + cn^3 & \text{Khi } n > 1 \end{cases}$$

Áp dụng định lý Master thu gọn với  $a = 3, b = 4$  và  $k = 3$  ta có  $a < b^k$  rơi vào trường hợp 3 nên  $T(n) = \Theta(n^3)$ .

**Ví dụ 2:** Xác định độ phức tạp của biểu thức

$$T(n) = \begin{cases} \Theta(1) & \text{Khi } n = 0, 1 \\ T\left(\frac{2n}{3}\right) + \Theta(1) & \text{Khi } n > 1 \end{cases}$$

Áp dụng định lý Master thu gọn với  $a = 1, b = 3/2$  và  $k = 0$  ta có  $a = b^k$  rơi vào trường hợp 2 nên  $T(n) = \Theta(\log n)$ .

**Ví dụ 3:** Xác định độ phức tạp của biểu thức

$$T(n) = \begin{cases} \Theta(1) & \text{Khi } n = 0, 1 \\ 2T\left(\frac{n}{2}\right) + cn & \text{Khi } n > 1 \end{cases}$$

Áp dụng định lý Master thu gọn với  $a = 2$ ,  $b = 2$  và  $k = 1$  ta có  $a = b^k$  rơi vào trường hợp 2 nên  $T(n) = \Theta(n \log n)$ .

#### **1.4. Bài tập**

Trong phần này bạn hãy lập trình các bài sau và đánh giá độ phức tạp thuật toán do bạn viết.

- Bài 1.** Nhập vào một dãy số nguyên gồm  $n$  ( $1 \leq n \leq 10^5$ ) phần tử  $a_1, a_2, \dots, a_n$  (trong đó  $-10^6 \leq a_i \leq 10^6$ ) tìm ước chung lớn nhất của dãy số nguyên.
- Bài 2.** Nhập vào một dãy số nguyên gồm  $n$  ( $1 \leq n \leq 10^6$ ) phần tử  $a_1, a_2, \dots, a_n$  (trong đó  $-10^6 \leq a_i \leq 10^6$ ) hãy xóa đi một số nào đó để ước chung lớn nhất của các số còn lại là lớn nhất, xuất ra giá trị ước chung lớn nhất tìm đạt được.
- Bài 3.** Nhập vào ba số thực  $a, b, c$  biện luận về số nghiệm của phương trình trùng phương  $ax^4 + bx^2 + c = 0$ .
- Bài 4.** Đếm số nguyên tố trong đoạn  $[L, R]$  cho trước. Nhập vào  $n$  ( $1 \leq n \leq 10^5$ ) là số lần đoạn cần đếm, với mỗi đoạn gồm hai số nguyên dương ( $1 \leq L_i \leq R_i \leq 10^5$ ) hãy đếm số nguyên tố trong đoạn từng đoạn và xuất  $n$  kết quả tìm được ra màn hình.
- Bài 5.** Theo giả thiết Goldbach “mọi số nguyên dương lớn hơn hay bằng 6 đều là tổng của ba số nguyên tố”. Nhập vào số nguyên dương  $n$  ( $1 \leq n \leq 10^5$ ) hãy đếm số bộ 3 số nguyên tố  $p_1 \leq p_2 \leq p_3$  có tổng bằng  $n$ .
- Bài 6.** Nhập vào một dãy số nguyên gồm  $n$  ( $1 \leq n \leq 10^5$ ) phần tử  $a_1, a_2, \dots, a_n$  (trong đó  $-10^6 \leq a_i \leq 10^6$ ) và một giá trị  $T$  ( $-10^9 \leq a_i \leq 10^9$ ) hãy tìm dãy con liên tục dài nhất có tổng bằng  $T$  và nếu tồn tại thì xuất ra hai chỉ số đầu và cuối nhỏ nhất tìm được, ngược lại xuất ra giá trị -1.

- Bài 7.** Nhập vào một dãy số nguyên gồm  $n$  ( $1 \leq n \leq 10^5$ ) phần tử đã được sắp xếp không giảm  $a_1 \leq a_2 \leq a_3 \leq \dots \leq a_n$ , (trong đó  $-10^6 \leq a_i \leq 10^6$ ) và một giá trị  $k$  ( $-10^6 \leq k \leq 10^6$ ) hãy chỉ ra một giá trị trong dãy lớn nhất không vượt quá  $k$ , trong trường hợp không có phần tử nào thỏa mãn xuất ra màn hình “khong co phan tu thoa man”.
- Bài 8.** Số nguyên dương được gọi là số tựa nguyên tố nếu nó có đúng 3 ước số nguyên dương khác nhau. Bạn hãy nhập vào số nguyên dương  $n$  với  $1 \leq n \leq 10^{18}$ , kiểm tra xem số  $n$  có tựa nguyên tố không?
- Bài 9.** Nhập vào một dãy số nguyên gồm  $n$  ( $1 \leq n \leq 10^5$ ) phần tử  $a_1, a_2, \dots, a_n$  (trong đó  $-10^6 \leq a_i \leq 10^6$ ) hãy tìm một dãy con bitonic dài nhất đó là một dãy không tăng hoặc không giảm hoặc nửa đầu không giảm và nửa cuối không tăng.
- Bài 10.** Nhập vào một dãy số nguyên gồm  $n$  ( $1 \leq n \leq 10^5$ ) phần tử  $a_1, a_2, \dots, a_n$  (trong đó  $-10^6 \leq a_i \leq 10^6$ ) hãy tìm giá trị nhỏ nhất của tích hai số bất kỳ trong dãy.
- Bài 11.** Nhập vào một dãy số nguyên gồm  $n$  ( $1 \leq n \leq 10^5$ ) phần tử  $a_1, a_2, \dots, a_n$  (trong đó  $-10^6 \leq a_i \leq 10^6$ ) hãy tìm giá trị nhỏ nhất của tích hai số bất kỳ trong dãy.
- Bài 12.** Nhập vào một ma trận các số chữ số có  $n$  hàng và  $m$  cột  $(a_{ij})_{n \times m}$  trong đó  $1 \leq n, m \leq 100$  và  $a_{ij}$  là các ký tự in thường Tiếng Anh thuộc tập ‘a’... ‘z’, một góc quay là bội của  $90^\circ$  hãy thực hiện phép quay ma trận xuôi chiều kim đồng hồ theo góc quay đó.
- Bài 13.** Viết các thuật toán sắp xếp nổi bọt, chọn, chèn bằng đệ quy và đánh giá độ phức tạp của các thuật toán đó.

## **Chương 2. Phân tích thiết kế thuật toán lặp và đệ quy**

Chúng ta biết rằng, khi giải các bài toán toán học chúng ta quan tâm tới đáp số của bài toán thì khi giải các bài toán bằng lập trình chúng ta lại quan tâm tới con đường đi đến đáp số của bài toán. Trong toán học sự đa dạng của các loại bài toán từ kết quả là đúng hay sai có hay không, những bài toán đếm

là có bao nhiêu kết quả, cho đến những bài toán liệt kê tất cả các kết quả giải được, thì trong lập trình số các con đường đi đến kết quả còn phong phú hơn nhiều có thể là những con đường trực tiếp như mô tả bài toán, có những con đường sử dụng các kết quả định lý toán học đã được chứng minh, có những con đường đi từ trên xuống hoặc từ dưới lên và vô số các con đường đi khác nhau. Hơn nữa, khác với toán học cần những kết quả chính xác thì nhiều khi với những kết quả xấp xỉ nhưng tìm ra trong thời gian cho phép đã là con đường chấp nhận được để lập trình. Mặt khác khi lập trình có những con đường bằng phẳng với những cấu trúc dữ liệu đơn sơ, lại có những con đường sử dụng các mô hình, cấu trúc dữ liệu phức tạp nhưng hiệu quả về thời gian.

### 2.1. Phân tích thiết kế thuật toán

Chúng ta muốn tìm thuật toán để giải quyết bài toán thì trước hết chúng ta phải phân tích thuật toán, sau đó chọn cấu trúc dữ liệu phù hợp, thiết kế thuật giải, chứng minh tính đúng đắn của thuật toán, chọn ngôn ngữ lập trình, cài đặt thuật toán, kiểm thử và gỡ lỗi logic để hoàn thiện chương trình.

#### 2.1.1. Các yếu tố phân tích, thiết kế thuật toán

Khi phân tích, thiết kế thuật toán ta phân tích, thiết kế các yếu tố:

- a. **Đặc tả bài toán:** Phân tích bài toán là bước đầu tiên và quan trọng nhất, chúng ta phải hiểu được bài toán yêu cầu làm gì, xác định các điều kiện và miền ràng buộc của dữ liệu vào (tiền điều kiện) và dữ liệu ra (hậu điều kiện), hiểu được các ví dụ mẫu nếu có.
- b. **Các thức tiến hành (ý tưởng giải quyết bài toán):** Sau khi đặc tả bài toán cho chúng ta những ý tưởng khác nhau để giải quyết bài toán, chúng ta lựa chọn ý tưởng phù hợp nhất cho bài toán đặt ra và mô hình hóa bởi sơ đồ thuật giải hoặc giả mã thậm chí mã thật cho thuật toán.
- c. **Tính đúng đắn của thuật toán:** Đối với thuật toán lập, tính đúng đắn của thuật toán được chứng minh thông qua xác định bất biến vòng lặp, đó là một mệnh đề logic đúng trước và sau mỗi bước lặp. Để đảm bảo

vòng lặp hoạt động đúng thì thiết lập mệnh đề logic phải đúng trước khi vào vòng lặp, trong khi lặp và sau khi lặp gồm:

- Thiết lập bất biến trước khi vào vòng lặp;
- Duy trì bất biến trong từng bước trong vòng lặp;
- Đảm bảo bất biến đúng khi ra khỏi vòng lặp.

Cần chú ý, trong khi thực hiện vòng lặp đôi khi có thể có những tình huống thoát bất thường khỏi vòng lặp do tìm thấy kết quả hoặc có những bất thường không tìm được kết quả đối với những dữ liệu đầu vào đặc biệt nào đó.

**d. Truy hồi và đệ quy:** Khi xác định được bất biến vòng lặp ta xây dựng biểu thức toán học truy hồi và chương trình đệ quy tương ứng để cài đặt thuật toán.

**e. Độ phức tạp của thuật toán:** Độ phức tạp thuật toán lặp và đệ quy sẽ cho chúng ta thấy được tốc độ tính toán và so sánh tốt xấu so với các thuật toán khác.

### **2.1.2. Một số ví dụ**

**Ví dụ 1:** Nhập vào số nguyên không âm  $n$  và tính  $n!$

#### **a. Đặc tả bài toán**

- **Tiền điều kiện:** Dữ liệu vào là số nguyên không âm  $n$ .
- **Hậu điều kiện:** Dữ liệu ra  $n! = 1 \times 2 \times \dots \times n$  cũng là số nguyên dương.

#### **b. Các thức tiến hành (ý tưởng thuật toán)**

Ta đặt  $S_n = n! = 1 \times 2 \times \dots \times n$ , để thấy  $S_i = i! = 1 \times 2 \times \dots \times (i-1) \times i = S_{i-1} \times i$  còn  $S_0 = 0! = 1$ . Như vậy để tính  $n!$  ta sẽ sử dụng một mảng dữ liệu  $S[1000]$ , với xuất phát khởi gán  $S[0] = 1$ , sau đó lần lượt đi tính các giá trị  $S[1] = S[0] * 1$ ;  $S[2] = S[1] * 2$ ; ...  $S[i] = S[i-1] * i$ ; ... ;  $S[n] = S[n-1] * n$  theo các bước sau:

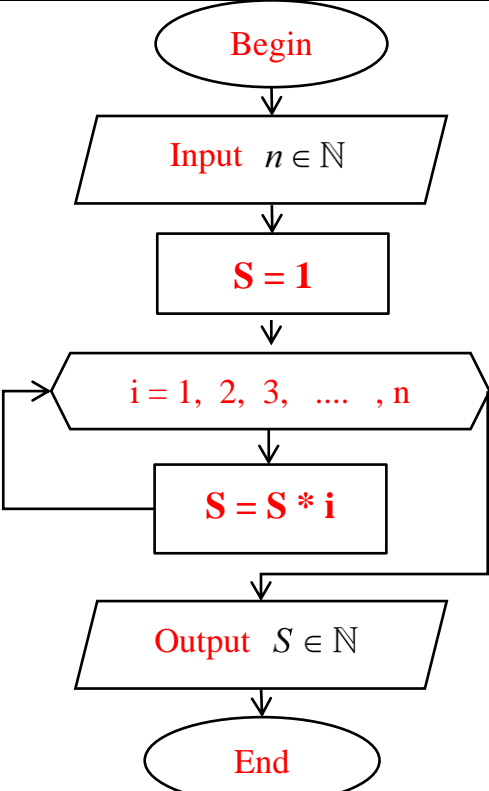
Bước 1. Nhập vào giá trị nguyên không âm  $n$ .

Bước 2. Khởi gán  $S[0] = 1$ .

Bước 3. Thực hiện vòng lặp  $i$  lần lượt từ 1 đến  $n$  với mỗi bước lặp ta tính  $S[i] = S[i-1] * i$ .

Bước 4. Xuất ra giá trị  $S[n]$  là kết quả cần tính.

Thông thường người ta hay tối ưu độ phức tạp không gian thuật toán bằng cách thay vì dùng mảng  $S[1000]$  thì ta dùng một biến  $S$  theo sơ đồ thuật toán sau:

Sơ đồ	Chương trình
	<pre>#include&lt;stdio.h&gt; int main() {     int n;     printf("Nhập vào n = ");     scanf("%d",&amp;n);     long long S=1;     for(int i=1;i&lt;=n;i++)         S*=i;     printf("S = %lld",S); }</pre>

**Chú ý:**

- Chúng ta có thể cho vòng lặp chạy từ 2 đến  $n$  thì thuật toán vẫn đúng;
- Nếu chúng ta khởi gán  $S = 3! = 6$  và cho vòng lặp chạy từ 4 đến  $n$  thì thuật toán sẽ đúng với  $n > 2$  còn với  $n < 2$  thì đều có kết quả sai là 6.
- Chúng ta cũng có thể cho  $i$  chạy ngược từ  $n$  về 1 hoặc sử dụng vòng lặp while, do – while hoặc dùng lệnh nhảy goto thay cho vòng for cũng được.

### c. Tính đúng đắn của thuật toán

**Bất biến vòng lặp:** Mỗi bước thứ  $i$  của vòng lặp đều có  $S_i = i! = 1 \times 2 \times \dots \times i$

với mọi giá trị của  $i$  từ 1 đến  $n$ . Ta có thể biểu diễn bằng mệnh đề logic toán  $(1 \leq i) \wedge (i \leq n) \wedge (S_i = i!)$  hoặc biểu diễn dưới dạng logic Hoare của chương trình trên là  $\{S = 1\}$  **for**(**int**  $i=1; i \leq n; i++$ )  $S*=i$   $\{S = n!\}$ , trong đó  $\{S = 1\}$  là điều kiện trước khi vào vòng lặp, còn  $\{S = n!\}$  là điều kiện kết thúc khi ra khỏi vòng lặp.

- **Thiết lập bất biến vòng lặp:** Trước khi vào vòng lặp  $S = 1$ , điều này rất quan trọng bởi vì nếu ta gán cho  $S$  bất kỳ giá trị nào khác thì khi ra khỏi vòng lặp đều không thu được  $n!$ .

- **Duy trì bất biến vòng lặp:** Khi ở bước thứ  $i - 1$  thì  $S = (i - 1)!$ , với mong muốn đạt được sang bước thứ  $i$  là  $S = i!$ . Như vậy để duy trì bất biến thì  $S = S*i$ .

**Thoát khỏi vòng lặp:** Không có tình huống nào thoát bất thường, chỉ có thoát bình thường khỏi vòng lặp ta thu được sau  $n$  bước là  $S = n!$ , đảm bảo tính đúng đắn của thuật toán.

### d. Truy hồi và đệ quy

Theo phân tích bất biến vòng lặp ta có biểu thức truy hồi

$$S_n = \begin{cases} 1 & \text{Khi } n = 0 \\ S_{n-1} * n & \text{Khi } n > 0 \end{cases}$$

Từ biểu thức truy hồi ta xây dựng hàm đệ quy tính giai thừa như sau:

```
#include<stdio.h>
long S(int n)
{
    if(n == 0) return 1;
    return n * S(n-1);
}
int main()
{
    int n;
    printf("Nhập vào n = ");scanf("%d",&n);
    printf("\nn! = %lld",n,S(n));
}
```



}

### e. Độ phức tạp của thuật toán

- Độ phức tạp của thuật toán lặp là  $O(n)$  do chỉ có một vòng lặp  $n$  bước
- Độ phức tạp của thuật toán đệ quy cũng là  $O(n)$  (*xem mục 1.3.4*)

**Ví dụ 2:** Nhập vào số nguyên dương  $n > 1$ , kiểm tra  $n$  có là số nguyên tố không?

### a. Đặc tả bài toán

- **Tiền điều kiện:** Dữ liệu vào là số nguyên dương  $n$  ( $n \in \mathbb{N}, n > 1$ ).
- **Hậu điều kiện:** Kết luận có hoặc không nguyên tố là một kiểu dữ liệu **bool**.

### b. Các thức tiến hành (ý tưởng thuật toán)

Số nguyên dương  $n > 1$  có là số nguyên tố khi và chỉ khi nó không chia hết cho bất kỳ số nguyên dương nào thuộc đoạn  $[2, \lfloor \sqrt{n} \rfloor]$  (**xem ví dụ trong mục 1.3.2**). Đặt  $m = \lfloor \sqrt{n} \rfloor$ , để kiểm tra  $n$  có là số nguyên tố hay không ta kiểm tra xem  $n$  có chia hết cho ít nhất 1 số nguyên nào đó thuộc đoạn  $[2, m]$  hay không theo các bước sau:

Bước 1. Nhập vào giá trị nguyên  $n > 1$ .

Bước 2. Đặt  $m = \lfloor \sqrt{n} \rfloor$

Bước 3. Thực hiện vòng lặp  $i$  lần lượt từ 2 đến  $m$  với mỗi bước lặp ta kiểm tra xem nếu  $n$  chia hết cho  $i$  thì dừng lại và khẳng định  $n$  không nguyên tố.

Bước 4. Ra khỏi vòng lặp  $n$  không chia hết cho bất cứ số nào thì  $n$  nguyên tố.

### Chú ý:

- Nếu ta không tính  $m$  trước mà cho trực tiếp vào trong vòng lặp như sau:

`for(LL i=2; i<=sqrt(n); i++)` thì mỗi bước lặp chương trình sẽ gọi một lần hàm tính căn nên sẽ làm chậm tốc độ của thuật toán.

- Chúng ta có thể không cần tính căn mà thay bằng lệnh kiểm tra bình phương của  $i$  không vượt quá  $n$  `for(LL i=2; i*i<=n; i++)` thì không ảnh hưởng tới độ phức tạp của thuật toán.

Sơ đồ	Chương trình
<pre> graph TD     Begin([Begin]) --&gt; Input[/Input n ∈ ℕ, n &gt; 1/]     Input --&gt; M["m = ⌊√n⌋"]     M --&gt; Loop["i = 2, 3, ..., m"]     Loop --&gt; Decision{"n mod i = 0"}     Decision -- "+" --&gt; OutputFalse[/Output false/]     Decision -- "-" --&gt; OutputTrue[/Output true/]     OutputTrue --&gt; End([End])     </pre>	<pre> #include&lt;stdio.h&gt; #include&lt;math.h&gt; typedef unsigned long long LL;  bool ktnt(LL n) {     LL m=sqrt(n);     for(LL i=2; i&lt;=m; i++)         if(n%i==0) return false;     return true; }  int main() {     LL n;     printf("Nhap vao n = ");     scanf("%llu",&amp;n);     printf(ktnt(n)?"nguyen to":"khong nguyen to"); } </pre>

### c. Tính đúng đắn của thuật toán

**Bất biến vòng lặp:** Mỗi bước thứ  $i$  của vòng lặp đều có  $n$  không chia hết cho bất kỳ số nào từ 2 đến  $i$  với mọi giá trị của  $i$  từ 2 đến  $m$ . Ta có thể biểu diễn bằng mệnh đề logic toán  $(1 \leq i) \wedge (i \leq m) \wedge (n \bmod i \neq 0)$ .

- **Thiết lập bất biến vòng lặp:** Trước khi vào vòng lặp chưa kiểm tra bất kỳ tính chất chia hết cho số nào nên nó là **true**, điều này rất quan trọng bởi vì nếu ngay từ đầu đã **false** thì kết quả sẽ luôn nhận được giá trị **false**.

- **Duy trì bất biến vòng lặp:** Khi ở bước thứ  $i - 1$  thì  $n$  đã không chia hết cho bất kỳ số nào từ 2 đến  $i - 1$ . Sang đến bước thứ  $i$  có hai tình huống xảy ra nếu  $n$  chia hết cho  $i$  ta không chạy vòng lặp nữa mà dừng lại kết luận  $n$  không nguyên tố, ngược lại nếu  $n$  không chia hết cho  $i$  ta duy trì được bất biến vòng lặp để thực hiện tiếp bước lặp tiếp theo.

**Thoát khỏi vòng lặp:** Thuật toán thoát bất thường khi  $n$  chia hết cho một số  $i$  nào đó và như vậy  $n$  không phải là số nguyên tố, thoát bình thường khỏi vòng lặp ta thu được bất biến sau  $m$  bước vẫn thỏa mãn tính đúng do đó  $n$  là số nguyên tố.

#### **d. Truy hồi và đệ quy**

Đặt  $B(n,i)$  là bất biến vòng lặp ta có biểu thức truy hồi sau

$$B(n,i) = \begin{cases} \text{true} & \text{Khi } i < 2 \\ B(n, i-1) \wedge (n \bmod i \neq 0) & \text{Khi } 2 \leq i \leq \lfloor \sqrt{n} \rfloor \end{cases}$$

#### **Chương trình đệ quy xuôi**

```
#include<stdio.h>
#include<math.h>
typedef unsigned long long LL;
bool ktnt(LL n,LL i,LL m){
    if (i>m) return true;
    if(n%i==0) return false;
    return ktnt(n,i+1,m);
}
int main(){
    LL n,m;
    printf("Nhap vao n = "); scanf("%llu",&n); m=sqrt(n);
    printf(ktnt(n,2,m)?"nguyen to":"khong nguyen to");
}
```

#### **Chương trình đệ quy ngược**

```
#include<stdio.h>
#include<math.h>
typedef unsigned long long LL;
```

```
bool ktnt(LL n, LL i)
{
    if (i<2) return true;
    if(n%i==0) return false;
    return ktnt(n,i-1);
}
int main()
{
    LL n;
    printf("Nhap vao n = "); scanf("%llu",&n);
    LL m=sqrt(n);
    printf(ktnt(n,m)?"nguyen to":"khong nguyen to");
}
```

***e. Độ phức tạp của thuật toán***

Đối với thuật toán lặp và cả thuật toán đệ quy trong trường hợp tốt nhất vừa vào kiểm tra  $n$  đã chia hết cho 2 luôn chứng tỏ độ phức tạp có cận dưới  $\Omega(1)$ . Trong trường hợp xấu nhất  $n$  không chia hết cho bất cứ số nào mỗi bước lặp tuyến tính đi hết  $m - 1$  bước do đó độ phức tạp có cận trên là  $O(\sqrt{n})$

**2.2. Phân tích thiết kế thuật toán tính tổng**

**Ví dụ:** Nhập số thực  $x$  và số nguyên dương  $n$  tính giá trị biểu thức

$$T = \sqrt{\sin x - \frac{\cos x}{1} + \frac{\cos^3 x}{3} - \frac{\cos^5 x}{5} + \frac{\cos^7 x}{7} - \dots + (-1)^{n+1} \frac{\cos^{2n+1} x}{2n+1}}$$

***a. Đặc tả bài toán***

- **Tiền điều kiện:** Dữ liệu vào là số thực  $x$  và số nguyên dương  $n$ .
- **Hậu điều kiện:** Dữ liệu ra hoặc là tính được  $S$  là một số thực không âm hoặc là không tính được giá trị của biểu thức.

***b. Các thức tiến hành (ý tưởng thuật toán)***

Ta có biểu thức cần tính

$$T = \sqrt{\sin x + \sum_{i=0}^n (-1)^{i+1} \frac{\cos^{2i+1} x}{2i+1}} = \sqrt{\sin x - \cos x \sum_{i=0}^n (-1)^i \frac{\cos^{2i} x}{2i+1}}$$

$$\text{Đặt } t = -\cos^2 x \text{ và xét } S = \sum_{i=0}^n (-1)^i \frac{\cos^{2i} x}{2i+1} = \sum_{i=0}^n \frac{(-\cos^2 x)^i}{2i+1} = \sum_{i=0}^n \frac{t^i}{2i+1}$$

khi đó bài toán đưa về tìm giá trị của biểu thức S từ đó suy ra giá trị của  $T = \sqrt{\sin x - \cos x \times S}$ . Để thuận tiện cho trình bày ý tưởng ta xét hai biểu thức

$$S_n = \sum_{i=0}^n \frac{t^i}{2i+1} \text{ và } P_n = t^n \text{ thực hiện lần lượt các bước để tính giá trị của } S = S_n$$

.

Bước 1. Nhập vào giá trị thực x và số nguyên dương n

Bước 2. Đặt  $t = -\cos^2 x$

Bước 3. Thực hiện vòng lặp lần lượt để tính  $P_n$  và  $S_n$ .

$$\text{Bước 3.0: Khởi tạo } \begin{cases} P_0 = t^0 = 1 \\ S_0 = \frac{1}{1} = 1 \end{cases}$$

$$\text{Bước 3.1: Tính } \begin{cases} P_1 = t^1 = P_0 \times t \\ S_1 = \frac{1}{1} + \frac{t^1}{3} = S_0 + \frac{P_1}{3} \end{cases}$$

...

$$\begin{aligned} &\text{Bước 3.i: Tính} \\ &\begin{cases} P_i = t^i = P_{i-1} \times t \\ S_i = \frac{1}{1} + \frac{t^1}{3} + \frac{t^2}{5} + \dots + \frac{t^{i-1}}{2(i-1)+1} + \frac{t^i}{2i+1} = S_{i-1} + \frac{P_i}{2i+1} \end{cases} \end{aligned}$$

$$\text{Bước 3.n: Tính } \begin{cases} P_n = t^n = P_{n-1} \times t \\ S_n = \frac{1}{1} + \frac{t^1}{3} + \dots + \frac{t^{n-1}}{2(n-1)+1} + \frac{t^n}{2n+1} = S_{n-1} + \frac{P_n}{2n+1} \end{cases}$$

Bước 4. Tính  $S = \sin x - \cos x \times S_n$ .

Bước 5. Kiểm tra nếu  $S < 0$  thì không tính được ngược lại xuất  $\sqrt{S}$ .

Từ ý tưởng thuật toán để đơn giản ta không dùng mảng P[1000] và S[1000] mà dùng biến P và S, chúng ta thiết kế sơ đồ và cài đặt giải thuật như sau:

Sơ đồ	Chương trình
<pre> graph TD     Begin([Begin]) --&gt; Input[/Input <math>x \in \mathbb{R}, n \in \mathbb{N}^+</math>/]     Input --&gt; Init["<math>t = -\cos^2 x</math> <math>P = S = 1</math>"]     Init --&gt; LoopStart[/i = 1, 2, 3, ..., n/]     LoopStart --&gt; LoopBody["<math>P = P * t</math> <math>S = S + P/(2i+1)</math>"]     LoopBody --&gt; LoopStart     LoopBody --&gt; CalcS["<math>S = \sin x - \cos x * S</math>"]     CalcS --&gt; Decision{S &lt; 0}     Decision -- + --&gt; Out1[/Out khong tinh duoc/]     Decision -- - --&gt; Out2[/Out <math>\sqrt{S}</math>/]     Out1 --&gt; End([End])     Out2 --&gt; End     </pre>	<pre> #include&lt;stdio.h&gt; #include&lt;math.h&gt; int main() {     float x,S,P,t;     int n,i;     printf("Nhap x = ");     scanf("%f",&amp;x);     printf("Nhap n = ");     scanf("%d",&amp;n);     t=cos(x);     t=-t*t;     P=S=1;     for(i=1;i&lt;=n;i++)     {         P*=t;         S+=P/(2*i+1);     }     S=sin(x)-cos(x)*S;     if(S&lt;0)printf("Khong tinh duoc T");     else printf("Gia tri T = %f",sqrt(S)); }     </pre>

**c. Tính đúng đắn của thuật toán**

**Bất biến vòng lặp:** Mỗi bước thứ  $i$  ( $1 \leq i \leq n$ ) của vòng lặp đều đại lượng bất biến  $B_i$  gồm hai thành phần  $P_i, S_i$  với công thức như sau:

$$B_i : \begin{cases} P_i = t^i \\ S_i = \frac{1}{1} + \frac{t^1}{3} + \frac{t^2}{5} + \dots + \frac{t^{i-1}}{2(i-1)+1} + \frac{t^i}{2i+1} \end{cases}$$

Ta có thể biểu diễn bằng mệnh đề logic toán  $(1 \leq i) \wedge (i \leq n) \wedge B_i$ .

- **Thiết lập bất biến vòng lặp:** Trước khi vào vòng lặp ở bước khởi tạo ta thiết lập bất biến  $P_0 = t^0 = 1$  và  $S_0 = \frac{1}{1} = 1$ , điều này rất quan trọng bởi vì nếu ngay từ đầu đã  $P_0$  hoặc  $S_0$  khác 1 thì kết quả tính sau cùng sẽ sai hết.

- **Duy trì bất biến vòng lặp:** Khi ở bước thứ  $i - 1$  thì bất biến sẽ là

$$B_{i-1} : \begin{cases} P_{i-1} = t^{i-1} \\ S_i = \frac{1}{1} + \frac{t^1}{3} + \frac{t^2}{5} + \dots + \frac{t^{i-1}}{2(i-1)+1} \end{cases}$$

Với mong muốn đạt được bất biến ở bước thứ  $i$  là

$$B_i : \begin{cases} P_i = t^i \\ S_i = \frac{1}{1} + \frac{t^1}{3} + \frac{t^2}{5} + \dots + \frac{t^{i-1}}{2(i-1)+1} + \frac{t^i}{2i+1} \end{cases}$$

ta phải duy trì bất biến bằng cách tính  $P_i = P_{i-1} \times t$  còn  $S_i = S_{i-1} + P_i / (2i+1)$ .

**Thoát khỏi vòng lặp:** Thuật toán không có thoát bất thường bởi mọi phép tính để duy trì bất biến không gây ra lỗi toán học nào bởi phép chia không có khả năng chia cho 0, thoát bình thường khỏi vòng lặp ta thu được bất biến sau  $n$  bước  $B_n$  gồm  $P_n$  và  $S_n$  như sau:

$$B_n : \begin{cases} P_n = t^n \\ S_n = \frac{1}{1} + \frac{t^1}{3} + \dots + \frac{t^{n-1}}{2(n-1)+1} + \frac{t^n}{2n+1} \end{cases}$$

**d. Truy hồi và đệ quy**

Đại lượng truy hồi  $B_n$  gồm hai thành phần  $P_n$  và  $S_n$  là bất biến vòng lặp ta được biểu diễn dưới dạng biểu thức truy hồi:

$$B_n : \begin{cases} \begin{cases} P_n = t^n = 1 \\ S_n = \frac{1}{1} = 1 \end{cases} & \text{Khi } n=0 \\ \begin{cases} P_n = P_{n-1} \times t \\ S_n = S_{n-1} + P_n / (2n+1) \end{cases} & \text{Khi } n > 0 \end{cases}$$

Chương trình đệ quy xây dựng dựa trên biểu thức truy hồi

```
#include <stdio.h>
#include <math.h>
void TinhPS(float t, int n, float &P, float &S)
{
    if(n==0) {P=S=1; return;}
    TinhPS(t,n-1,P,S);
    P = P * t;
    S = S + P/(2*n+1);
}
int main()
{
    int n;
    float x,P,S,t;
    printf("Nhap x va n");
    scanf("%f%d",&x,&n);
    t=cos(x);
    t=-t*t;
    TinhPS(t,n,P,S);
    S=sin(x)-cos(x)*S;
    if(S<0)printf("Khong tinh duoc T");
    else printf("Gia tri T = %f",sqrt(S));
}
```



***e. Độ phức tạp của thuật toán***

Đối với thuật toán lặp vòng lặp không có thoát bất thường và duyệt tuyến tính gồm  $n$  bước nên độ phức tạp  $\Theta(n)$ .

Đối với thuật toán đệ quy ta gọi độ phức tạp của thuật toán là  $T_n$  xét trên chương trình đệ quy ta thấy rằng

$$T(n) = \begin{cases} c_0 & \text{Khi } n = 0 \\ T(n-1) + c & \text{Khi } n > 0 \end{cases}$$

Khai triển theo công thức này ta có

$$\begin{aligned} T(n) &= T(n-1) + c \\ &= T(n-2) + c + c && \text{Thay } T(n-1) = T(n-2) + c \\ &= T(n-2) + 2c \\ &= T(n-3) + 3c \\ &= \dots \\ &= T(n-n) + nc = T(0) + nc = c_0 + nc \end{aligned}$$

Vậy độ phức tạp của thuật toán đệ quy là  $T(n) = \Theta(n)$ .

**2.3. Phân tích thiết kế thuật toán trên dãy****2.3.1. Tìm phần tử âm lớn nhất trong dãy**

**Bài toán:** Nhập vào một dãy số nguyên gồm  $n$  ( $1 \leq n \leq 10^5$ ) phần tử  $a_1, a_2, \dots, a_n$ , (trong đó  $-10^6 \leq a_i \leq 10^6$ ) hãy tìm giá trị âm lớn nhất trong dãy nếu có.

**a. Đặc tả bài toán**

- **Tiền điều kiện:** Dữ liệu vào là số nguyên  $n$  ( $1 \leq n \leq 10^5$ ) và dãy số nguyên  $a_1, a_2, \dots, a_n$  ( $-10^6 \leq a_i \leq 10^6$ ).
- **Hậu điều kiện:** Dữ liệu ra hoặc dãy không có phần tử âm hoặc là có, trong trường hợp có chỉ ra giá trị âm lớn nhất.

**b. Các thức tiến hành (ý tưởng thuật toán)**

Chúng ta thấy rằng đầu ra cần 2 thông tin, thứ nhất là dãy có phần tử âm hay không và thứ 2 là giá trị âm lớn nhất trong dãy nếu có. Tương ứng với hai thông tin này ta cần tìm hai đại lượng  $Q_i$  dữ liệu kiểu **bool** sẽ lưu thông tin

$a_1, a_2, \dots, a_i$  có phần tử nào âm không nếu có sẽ nhận giá trị **true** ngược lại nhận giá trị **false**;  $M_i$  dữ liệu kiểu nguyên sẽ lưu thông tin giá trị âm lớn nhất trong dãy  $a_1, a_2, \dots, a_i$  theo các bước sau:

Bước 1. Nhập số nguyên dương  $n$  ( $1 \leq n \leq 10^5$ ).

Bước 2. Nhập dãy số nguyên  $n$  phần tử  $a_1, a_2, \dots, a_n$  ( $-10^6 \leq a_i \leq 10^6$ ).

Bước 3. Thực hiện vòng lặp lần lượt để tính  $Q_n$  và  $M_n$ .

Bước 3.0. Khởi gán  $Q_0 = \text{false}$ , dãy không có phần tử nào nên chưa có phần tử âm ta không quan tâm tới giá trị của  $M_0$ .

Bước 3.1. Xét giá trị  $a_1 \geq 0$  thì  $M_1 = M_0$  và  $Q_1 = Q_0$  ngược lại thì  $Q_1$  nhận giá trị **true** và  $M_1 = a_1$ .

...

Bước 3.i. Xét giá trị  $a_i \geq 0$  thì  $M_i = M_{i-1}$  và  $Q_i = Q_{i-1}$  ngược lại  $a_i < 0$  khi đó:

- Tính  $M_i$ : nếu  $Q_{i-1} = \text{true}$  chứng tỏ  $a_1, a_2, \dots, a_{i-1}$  đã có giá trị âm và  $M_{i-1}$  lưu giá trị âm lớn nhất trong dãy  $a_1, a_2, \dots, a_{i-1}$  nên  $M_i = \max(M_{i-1}, a_i)$ , ngược lại  $Q_{i-1} = \text{false}$  thì  $a_1, a_2, \dots, a_{i-1}$  không có phần tử nào âm nên  $M_i = a_i$ .
- Tính  $Q_i$ : vì  $a_i < 0$  nên trong dãy  $a_1, a_2, \dots, a_i$  có ít nhất một phần tử âm, do đó  $Q_i = \text{true}$  mà không cần quan tâm tới giá trị của  $Q_{i-1}$ .

...

Bước 3.n. Xét giá trị  $a_n \geq 0$  thì  $M_n = M_{n-1}$  và  $Q_n = Q_{n-1}$  ngược lại  $a_n < 0$  khi đó:

- Tính  $M_n$ : nếu  $Q_{n-1} = \text{true}$  chứng tỏ  $a_1, a_2, \dots, a_{n-1}$  đã có giá trị âm và  $M_{n-1}$  lưu giá trị âm lớn nhất trong dãy  $a_1, a_2, \dots, a_{n-1}$  nên  $M_n = \max(M_{n-1}, a_n)$ , ngược lại  $Q_{n-1} = \text{false}$  thì  $a_1, a_2, \dots, a_{n-1}$  không có phần tử nào âm nên  $M_n = a_n$ .

- Tính  $Q_n$ : vì  $a_n < 0$  nên trong dãy  $a_1, a_2, \dots, a_{n-1}$  có ít nhất một phần tử âm, do đó  $Q_n = \text{true}$ .

Bước 4. Khi ra khỏi vòng lặp nếu  $Q_n = \text{true}$  thì giá trị âm lớn nhất của dãy sẽ là  $M_n$  ngược lại xuất ra thông báo dãy không có phần tử nào âm.

Từ ý tưởng thuật toán ta thiết kế sơ đồ và cài đặt giải thuật như sau:

Sơ đồ	Chương trình
<pre> graph TD     Begin([Begin]) --&gt; Input[/Input <math>n \in \mathbb{N}^+, \{a_i\}_{i=1}^n \in \mathbb{Z}</math>/]     Input --&gt; M0[M = 0]     M0 --&gt; LoopStart{i = 1, 2, 3, ..., n}     LoopStart --&gt; AiLt0{<math>a_i &lt; 0</math>}     AiLt0 -- + --&gt; MAssign[M = <math>a_i</math>]     AiLt0 -- - --&gt; MMax[M = max(M, <math>a_i</math>)]     MAssign --&gt; LoopEnd(( ))     MMax --&gt; LoopEnd     LoopEnd --&gt; MZero{M = 0}     MZero -- + --&gt; OutNoNeg[/Out không có pt am/]     MZero -- - --&gt; OutM[/Out M/]     OutNoNeg --&gt; End([End])     OutM --&gt; End     </pre>	<pre> #include&lt;stdio.h&gt; #include&lt;math.h&gt; #define Max(a,b) (a)&gt;(b)?(a):(b) int main() {     long i,n,a[100006],M;     printf("Nhap n: ");     scanf("%ld",&amp;n);     for(i=1;i&lt;=n;i++)     {         printf("a[%ld]: ",i);         scanf("%ld",&amp;a[i]);     }     M=0;     for(i=1;i&lt;=n;i++)     if(a[i]&lt;0)     M=M==0?a[i]:(Max(M,a[i]));     if(M==0)     printf("Day khong co phan     tu am");     else printf("Gia tri am     lon nhat %ld",M); }     </pre>

**Giải thích:** Để đơn giản ta không cần dùng mảng để lưu các giá trị  $Q_0, Q_1, \dots, Q_n$  mà dùng một biến  $Q$ , biến này sẽ nhận giá trị **true** từ khi dãy bắt

đầu xuất hiện phần tử âm đầu tiên còn trước đó thì nhận giá trị **false**; ta cũng không cần mảng lưu các giá trị  $M_0, M_1, \dots, M_n$  mà dùng một biến  $M$ , biến này sẽ lưu giá trị âm lớn nhất thu được từ khi biến  $Q$  bắt đầu nhận giá trị true còn trước đó giá trị của biến này không cần quan tâm. Tối ưu hơn nữa ta sẽ sử dụng biến  $M$  một cách tinh tế đóng vai trò cho cả  $Q$  và  $M$  bằng cách khởi gán bằng một giá trị không âm bất kỳ (có thể chọn luôn bằng 0) khi nào xuất hiện phần tử âm đầu tiên ( $M = 0$ ) khi đó  $M$  sẽ lưu giá trị lớn nhất của các phần tử âm còn nếu không vẫn giữ nguyên giá trị 0 và sơ đồ đã thiết kế theo cách này.

### ***c. Tính đúng đắn của thuật toán***

**Bất biến vòng lặp:** Mỗi bước thứ  $i$  ( $1 \leq i \leq n$ ) của vòng lặp đều có đại lượng bất biến  $M_i$  có giá trị bằng 0 nếu dãy  $a_1, a_2, \dots, a_i$  không có phần tử nào âm hoặc ngược lại nếu có ít nhất một phần tử âm thì  $M_i$  chính là giá trị âm lớn nhất trong dãy này.

- **Thiết lập bất biến vòng lặp:** Trước khi vào vòng lặp ở bước khởi tạo dãy chưa có phần tử nào và do đó chưa có phần tử âm nên ta thiết lập bất biến  $M_0 = 0$ .

- **Duy trì bất biến vòng lặp:** Khi ở bước thứ  $i - 1$  thì bất biến sẽ là

$$M_{i-1} = \begin{cases} 0 & \text{Khi } a_j \geq 0 \quad \forall j \in \{1, \dots, i-1\} \\ \max_{1 \leq j \leq i-1} (a_j < 0) & \text{Khi } a_j < 0 \quad \exists j \in \{1, \dots, i-1\} \end{cases}$$

với mong muốn đạt được bất biến ở bước thứ  $i$  là

$$M_i = \begin{cases} 0 & \text{Khi } a_j \geq 0 \quad \forall j \in \{1, \dots, i\} \\ \max_{1 \leq j \leq i} (a_j < 0) & \text{Khi } a_j < 0 \quad \exists j \in \{1, \dots, i\} \end{cases}$$

ta phải duy trì bất biến bằng cách nếu gặp  $a_i \geq 0$  thì  $M_i = M_{i-1}$  ngược lại nếu  $M_{i-1} = 0$  thì đây sẽ là phần tử âm đầu tiên nên  $M_i = a_i$  còn không thì  $M_i = \max(M_{i-1}, a_i)$  để thu được giá trị âm lớn nhất trong  $a_1, a_2, \dots, a_i$ .

**Thoát khỏi vòng lặp:** Thuật toán không có thoát bất thường bởi mọi phép tính để duy trì bất biến không gây ra lỗi toán học nào, thoát bình thường ở ta thu

được  $M_n$  nếu giá trị này bằng 0 thì dãy không có phần tử âm ngược lại ta xuất ra giá trị âm lớn nhất là  $M_n$ .

#### ***d. Truy hồi và đệ quy***

Đại lượng truy hồi  $M_n$  là bất biến vòng lặp ta được biểu diễn dưới dạng biểu thức truy hồi:

$$M_n = \begin{cases} 0 & \text{Khi } n=0 \\ M_{n-1} & \text{Khi } n > 0, a_n \geq 0 \\ a_n & \text{Khi } n > 0, a_n < 0, M_{n-1} = 0 \\ \text{Max}(M_{n-1}, a_n) & \text{Khi } n > 0, a_n < 0, M_{n-1} \neq 0 \end{cases}$$

Chương trình đệ quy xây dựng dựa trên biểu thức truy hồi

```
#include<stdio.h>
#include<math.h>
#define Max(a,b) (a)>(b)?(a):(b)
long MaxAm(long n,long *a)
{
    if(n==0) return 0;
    long M=MaxAm(n-1,a);
    if(a[n]>0) return M;
    if(M==0) return a[n];
    return Max(M,a[n]);
}
int main()
{
    long i,n,a[100006],M;
    printf("Nhap n: ");
    scanf("%ld",&n);
    for(i=1;i<=n;i++){
        printf("a[%ld]:",i);
        scanf("%ld",&a[i]);
    }
    M=MaxAm(n,a);
    if(M==0) printf("Day khong co phan tu am");
    else printf("Gia tri am lon nhat %ld",M);
}
```

**e. Độ phức tạp của thuật toán**

Đối với thuật toán lặp vòng lặp không có thoát bất thường và duyệt tuyến tính gồm  $n$  bước nên độ phức tạp  $\Theta(n)$ .

Đối với thuật toán đệ quy ta gọi độ phức tạp của thuật toán là  $T_n$  xét trên chương trình đệ quy ta thấy rằng:

$$T(n) = \begin{cases} c_0 & \text{Khi } n = 0 \\ T(n-1) + c & \text{Khi } n > 0 \end{cases}$$

Khai triển theo công thức này ta có

$$\begin{aligned} T(n) &= T(n-1) + c \\ &= T(n-2) + c + c && \text{Thay } T(n-1) = T(n-2) + c \\ &= T(n-2) + 2c \\ &\dots \\ &= T(n-n) + nc = T(0) + nc = c_0 + nc \end{aligned}$$

Vậy độ phức tạp của thuật toán đệ quy là  $T(n) = \Theta(n)$ .

**2.3.2. Đếm số đoạn cắt trục hoành**

**Bài toán:** Nhập vào một dãy điểm tọa độ thực trên trục tọa độ Đề-các (Descartes) hai chiều vuông góc gồm  $n$  ( $1 \leq n \leq 10^5$ ) điểm đôi một khác nhau  $A_1(x_1, y_1), A_2(x_2, y_2), \dots, A_n(x_n, y_n)$ , (trong đó  $-10^6 \leq x_i, y_i \leq 10^6$ ). Hãy đếm số đoạn có hai đầu mút tạo bởi cặp 2 điểm bất kỳ cắt (có duy nhất một điểm chung) với trục hoành (đường  $y = 0$ ).

**a. Đặc tả bài toán**

- **Tiền điều kiện:** Dữ liệu vào là số nguyên  $n$  ( $1 \leq n \leq 10^5$ ) và dãy điểm thực  $A_1(x_1, y_1), A_2(x_2, y_2), \dots, A_n(x_n, y_n)$  trong đó  $-10^6 \leq x_i, y_i \leq 10^6$
- **Hậu điều kiện:** Dữ liệu ra là số đoạn có 1 điểm chung duy nhất với trục hoành.

**b. Các thức tiến hành (ý tưởng thuật toán)**

Chúng ta thấy rằng: theo quy tắc nhân trong phương pháp đếm, thay vì đếm số đoạn cắt trục hoành ta đếm số điểm bên trên trục hoành và số điểm bên dưới trục hoành sẽ ra số đoạn cắt đứt hẳn, số điểm thuộc trục hoành với số

điểm không thuộc trục hoành sẽ ra số đoạn cắt và đầu mút thuộc luôn trục hoành. Chúng ta xét trong dãy  $n$  điểm  $A_1(x_1, y_1), A_2(x_2, y_2), \dots, A_n(x_n, y_n)$ , đặt số điểm nằm trên trục hoành là  $O_n$ , số điểm nằm dưới trục hoành là  $U_n$  và số điểm thuộc trục hoành là  $B_n$  thì kết quả cần tìm sẽ là  $O_n \times U_n + (O_n + U_n) \times B_n$

Như vậy bài toán đưa về tìm  $O_n, U_n, B_n$  theo các bước sau:

Bước 1. Nhập số nguyên dương  $n$  ( $1 \leq n \leq 10^5$ ).

Bước 2. Nhập dãy  $n$  điểm mỗi điểm hai tọa độ  $-10^6 \leq x_i, y_i \leq 10^6$ .

Bước 3. Thực hiện vòng lặp lần lượt để tính  $O_n, U_n, B_n$ .

Bước 3.0. Khởi gán  $O_0 = U_0 = B_0 = 0$ , dãy chưa có điểm nào nên chưa có điểm nào trên, dưới và thuộc trục hoành.

Bước 3.1. Xét điểm  $A_1(x_1, y_1)$ , tùy thuộc vào giá trị  $y_1$  dương thì điểm nằm phía trên, âm thì điểm nằm phía dưới, bằng 0 thì điểm thuộc trục hoành tức là:

+ Nếu  $y_1 > 0$  thì  $O_1 = O_0 + 1$  ngược lại thì  $O_1 = O_0$ ;

+ Nếu  $y_1 < 0$  thì  $U_1 = U_0 + 1$  ngược lại thì  $U_1 = U_0$ ;

+ Nếu  $y_1 = 0$  thì  $B_1 = B_0 + 1$  ngược lại thì  $B_1 = B_0$ .

...

Bước 3.i. Xét điểm  $A_i(x_i, y_i)$ , tùy thuộc vào giá trị  $y_i$  dương thì điểm nằm phía trên, âm thì điểm nằm phía dưới, bằng 0 thì điểm thuộc trục hoành tức là:

+ Nếu  $y_i > 0$  thì  $O_i = O_{i-1} + 1$  ngược lại thì  $O_i = O_{i-1}$ ;

+ Nếu  $y_i < 0$  thì  $U_i = U_{i-1} + 1$  ngược lại thì  $U_i = U_{i-1}$ ;

+ Nếu  $y_i = 0$  thì  $B_i = B_{i-1} + 1$  ngược lại thì  $B_i = B_{i-1}$ .

...

Bước 3.n. Xét điểm  $A_n(x_n, y_n)$ , tùy thuộc vào giá trị  $y_n$  dương thì điểm nằm phía trên, âm thì điểm nằm phía dưới, bằng 0 thì điểm thuộc trục hoành tức là:

- + Nếu  $y_n > 0$  thì  $O_n = O_{n-1} + 1$  ngược lại thì  $O_n = O_{n-1}$ ;
- + Nếu  $y_n < 0$  thì  $U_n = U_{n-1} + 1$  ngược lại thì  $U_n = U_{n-1}$ ;
- + Nếu  $y_n = 0$  thì  $B_n = B_{n-1} + 1$  ngược lại thì  $B_n = B_{n-1}$ .

Bước 4. Khi ra khỏi vòng lặp ta chỉ cần tính giá trị  $O_n \times U_n + (O_n + U_n) \times B_n$  sẽ ra số đoạn cắt trục hoành.

Khi chúng ta thiết kế thuật toán, để đơn giản thay vì dùng ba mảng tương ứng lưu  $O_0, O_1, \dots, O_n$ ,  $U_0, U_1, \dots, U_n$  và  $B_0, B_1, \dots, B_n$  ta sẽ dùng hai biến O, U để tính số điểm nằm trên và số điểm nằm dưới còn số điểm thuộc B = n – O – U nên kết quả sẽ là  $O \times U + (O + U) \times (n - O - U)$  theo sơ đồ và chương trình:

Sơ đồ	Chương trình
<pre> graph TD     Begin([Begin]) --&gt; Input[/Input <math>n \in \mathbb{N}^+, \{x_i, y_i\}_{i=1}^n \in \mathbb{R}</math>/]     Input --&gt; Init[O = U = 0]     Init --&gt; LoopStart{i = 1, 2, 3, ..., n}     LoopStart --&gt; Cond1{<math>y_i &gt; 0</math>}     Cond1 -- + --&gt; Oinc[O = O + 1]     Cond1 -- - --&gt; Cond2{<math>y_i &lt; 0</math>}     Cond2 -- + --&gt; Uinc[U = U + 1]     Oinc --&gt; LoopStart     Uinc --&gt; LoopStart     LoopStart --&gt; Output[/Output <math>O \times U + (O + U) \times (n - O - U)</math>/]     Output --&gt; End([End])         </pre>	<pre> #include&lt;stdio.h&gt; #include &lt;utility&gt; #define FOR(i,n) for(int i=1;i&lt;=n;i++) using namespace std; typedef pair&lt;float,float&gt;Diem; int main(){     long n,O=0,U=0;     float x,y;     printf("Nhap n: ");     scanf("%ld",&amp;n);     Diem *A=new Diem[n+5];     FOR(i,n){         printf("A[%d]: ",i);         scanf("%f%f",&amp;x,&amp;y);         A[i]=make_pair(x,y);     }     FOR(i,n){         O+=A[i].second&gt;0;         U+=A[i].second&lt;0;     }     printf("So doan cat la %d",O*U+(O+U)*(n-O-U));     delete []A; }         </pre>



**c. Tính đúng đắn của thuật toán**

**Bất biến vòng lặp:** Mỗi bước thứ  $i$  ( $1 \leq i \leq n$ ) của vòng lặp đều đại lượng bất biến  $O_i, U_i$  tương ứng là số những điểm nằm phía trên và phía dưới trục hoành (đường thẳng  $y = 0$ ).

- **Thiết lập bất biến vòng lặp:** Trước khi vào vòng lặp ở bước khởi tạo dãy chưa có điểm nào và do đó thiết lập bất biến  $O_0 = U_0 = 0$ .

- **Duy trì bất biến vòng lặp:** Khi ở bước thứ  $i - 1$  thì bất biến  $O_{i-1}$  là số điểm nằm phía trên đường  $y = 0$  trong dãy điểm  $A_1, A_2, \dots, A_{i-1}$  do đó nếu điểm thứ  $i$  là  $A_i(x_i, y_i)$  có tung độ dương thì thêm một điểm nữa nằm phía trên trục hoành nên  $O_i = O_{i-1} + 1$  ngược lại thì  $O_i = O_{i-1}$ . Tương tự như vậy nếu  $A_i(x_i, y_i)$  có tung độ âm thì thêm một điểm nằm phía dưới trục hoành nên  $U_i = U_{i-1} + 1$  và ngược lại  $U_i = U_{i-1}$ .

**Thoát khỏi vòng lặp:** Thuật toán không có thoát bất thường, khi ra khỏi vòng lặp ta tính được số điểm nằm trên trục hoành là  $O_n$  và số điểm nằm dưới trục hoành là  $U_n$  nên số điểm thuộc trục hoành sẽ là  $n - O_n - U_n$  vậy số đoạn cắt trục hoành sẽ là  $O_n + U_n + (O_n + U_n) \times (n - O_n - U_n)$ .

**d. Truy hồi và đệ quy**

Đại lượng truy hồi  $O_n, U_n$  là bất biến vòng lặp ta được biểu diễn dưới dạng biểu thức truy hồi:

$$(O_n, U_n) = \begin{cases} (0, 0) & \text{Khi } n=0 \\ (O_{n-1} + 1, U_{n-1}) & \text{Khi } n > 0, y_n > 0 \\ (O_{n-1}, U_{n-1} + 1) & \text{Khi } n > 0, y_n < 0 \\ (O_{n-1}, U_{n-1}) & \text{Khi } n = 0, y_n = 0 \end{cases}$$

Chương trình đệ quy xây dựng dựa trên biểu thức truy hồi

```
#include<stdio.h>
#include <utility>
#define FOR(i,n) for(int i=1;i<=n;i++)
using namespace std;
```

```
typedef pair<float,float> Diem;
void Dem(long n, Diem *A, long &O, long &U)
{
    if(n==0) {O=U=0;return;}
    Dem(n-1, A, O, U);
    O+=A[n].second>0;
    U+=A[n].second<0;
}
int main()
{
    long n, O, U;
    float x, y;
    printf("Nhap n: ");
    scanf("%ld", &n);
    Diem *A=new Diem[n+5];
    FOR(i, n)
    {
        printf("A[%d]: ", i);
        scanf("%f%f", &x, &y);
        A[i]=make_pair(x, y);
    }
    Dem(n, A, O, U);
    printf("So doan cat la %d", O*U+(O+U)*(n-O-U));
    delete []A;
}
```

**e. Độ phức tạp của thuật toán**

Đối với thuật toán lặp vòng lặp không có thoát bất thường và duyệt tuyến tính gồm  $n$  bước nên độ phức tạp  $\Theta(n)$ .

Đối với thuật toán đệ quy ta gọi độ phức tạp của thuật toán là  $T_n$  xét trên chương trình đệ quy ta thấy rằng:

$$T(n) = \begin{cases} c_0 & \text{Khi } n = 0 \\ T(n-1) + c & \text{Khi } n > 0 \end{cases}$$

Chúng ta có  $T(n) = T(n-1) + c = \dots = T(0) + nc = \Theta(n)$ . Vậy độ phức tạp của thuật toán đệ quy là  $T(n) = \Theta(n)$ .

**2.4. Bài tập**

Phân tích, thiết kế, chứng minh tính đúng đắn, cài đặt thuật toán lặp và thuật toán đệ quy cho các bài toán sau.

**Bài 1.** Nhập vào số nguyên dương  $n$  tính tổng tất cả các ước nguyên dương của  $n$ .

**Bài 2.** Nhập vào số nguyên dương  $n$  tính giá trị các biểu thức sau:

a.  $S_1 = \frac{1}{n} + \frac{2}{n-1} + \dots + \frac{n}{1}$

b.  $S_2 = \sqrt{n + \sqrt{n-1 + \sqrt{n-2 + \sqrt{\dots + \sqrt{1}}}}}$

c.  $S_3 = e + \frac{1}{1!} - \frac{1}{2!} + \dots + (-1)^{n+1} \frac{1}{n!}$

**Bài 3.** Nhập vào số nguyên dương  $n$  và số thực  $x$  tính giá trị các biểu thức sau nếu có:

a.  $S_4 = \frac{x}{1} - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \dots + (-1)^n \frac{x^{2n+1}}{2n+1}$

b.  $S_5 = \frac{\sin(1)}{x} + \frac{\sin(1+x)}{x(x+1)} + \frac{\sin(1+x+x^2)}{x(x+1)(x+2)} + \dots + \frac{\sin(1+x+\dots+x^n)}{x(x+1)(x+2)\dots(x+n)}$

c.  $S_6 = \sqrt{\frac{\cos(x^2+1) + \cos^2(x^2+1) + \dots + \cos^n(x^2+1)}{\sin(-x^2+x+1) - \sin^2(-x^2+x+1) + \dots + (-1)^{n+1} \sin^n(-x^2+x+1)}}$

**Bài 4.** Nhập vào số nguyên dương  $n$  và hai số thực  $x, y$  tính giá trị các biểu thức sau nếu có:

a.  $S_7 = \sqrt{2017 + \frac{\sin x + 1}{xy^2(y-1)} - \frac{\sin^2 x + 1}{xy^2(y^2-2)} + \dots + (-1)^n \frac{\sin^n x + 1}{xy^2(y^n-n)}}$

b.  $S_8 = \tan\left(\frac{1 \times y}{\tan(y-1)} + \frac{x \times y^2}{\tan(y-2)} + \dots + \frac{x^{n-1} \times y^n}{\tan(y-n)}\right)$

**Bài 5.** Nhập vào một dãy số nguyên gồm  $n$  ( $1 \leq n \leq 10^5$ ) phần tử  $a_1, a_2, \dots, a_n$  (trong đó  $-10^6 \leq a_i \leq 10^6$ ) tìm giá trị phần tử âm đầu tiên trong dãy.

**Bài 6.** Nhập vào một dãy số nguyên gồm  $n$  ( $1 \leq n \leq 10^5$ ) phần tử  $a_1, a_2, \dots, a_n$  (trong đó  $-10^6 \leq a_i \leq 10^6$ ) tìm giá trị lẻ nhỏ nhất trong dãy.

**Bài 7.** Nhập vào một dãy số nguyên gồm  $n$  ( $1 \leq n \leq 10^5$ ) phần tử  $a_1, a_2, \dots, a_n$  (trong đó  $-10^6 \leq a_i \leq 10^6$ ) sắp xếp chọn.

**Bài 8.** Nhập vào một dãy số nguyên gồm  $n$  ( $1 \leq n \leq 10^5$ ) phần tử  $a_1, a_2, \dots, a_n$  (trong đó  $-10^6 \leq a_i \leq 10^6$ ) sắp xếp chèn.

**Bài 9.** Nhập vào một dãy số thực gồm  $n$  ( $1 \leq n \leq 10^5$ ) phần tử  $a_1, a_2, \dots, a_n$  (trong đó  $-10^6 \leq a_i \leq 10^6$ ) tính giá trị các biểu thức sau nếu có:

a. 
$$S_9 = \frac{a_1}{a_1} + \frac{a_1 + (a_1 + a_2)}{a_1 \times (a_1 \times a_2)} + \dots + \frac{a_1 + (a_1 + a_2) + \dots + (a_1 + a_2 + \dots + a_n)}{a_1 \times (a_1 \times a_2) \times \dots \times (a_1 \times a_2 \times \dots \times a_n)}$$

b. 
$$S_{10} = \frac{\sqrt{a_1}}{\tan a_1} - \frac{\sqrt{a_2 + \sqrt{a_1}}}{\tan a_2} + \dots + (-1)^{n+1} \frac{\sqrt{a_n + \dots + \sqrt{a_2 + \sqrt{a_1}}}}{\tan a_n}$$

**Bài 10.** Nhập vào một dãy điểm tọa độ thực trên trục tọa độ Đề-các vuông góc hai chiều gồm  $n$  điểm đôi một khác nhau  $A_1(x_1, y_1) \dots A_n(x_n, y_n)$ . Tính độ dài đường gấp khúc đi qua tất cả các điểm lần lượt từ  $A_1$  đến  $A_n$

### **Chương 3. Phân tích thiết kế chia để trị**

#### **3.1. Phương pháp chia để trị (divide and conquer)**

Phương pháp chia để trị là một phương pháp thiết kế thuật toán theo phương pháp trên xuống (top-down) bằng cách đệ quy nhiều phân nhánh. Phương pháp chia để trị hoạt động bằng cách chia bài toán ban đầu thường có kích cỡ lớn thành nhiều bài toán nhỏ hơn cùng thể loại, cứ như vậy lặp lại nhiều lần, cho đến khi bài toán thu được đủ đơn giản có thể giải được trực tiếp. Sau đó lời giải của các bài toán nhỏ được tổng hợp lại thành lời giải của các bài toán to hơn cứ như vậy tới khi giải được bài toán ban đầu.

Cấu trúc chung của các thuật toán chia để trị có 3 phần gồm chia, trị và liên kết.

**Trường hợp bài toán suy biến:** Giải trực tiếp

**Trường hợp bài toán chưa suy biến:** thực hiện chia để trị

**Chia:** Chia bài toán lớn thành nhiều bài toán con kích thước nhỏ hơn nhưng có cùng cách giải;

**Trị:** Dùng đệ quy giải các bài toán con;

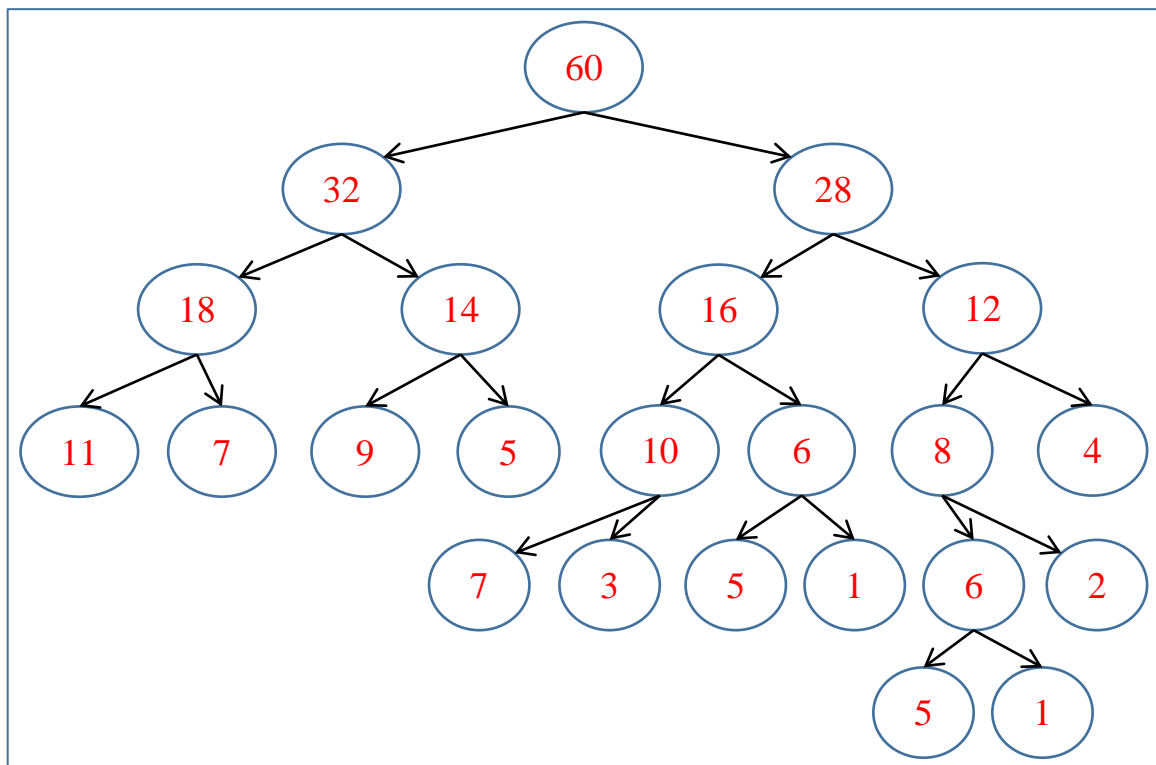
**Liên kết:** Gom kết quả của các bài toán con thành kết quả bài toán ban đầu.

#### **Ví dụ 1: Bài toán tách nhóm**

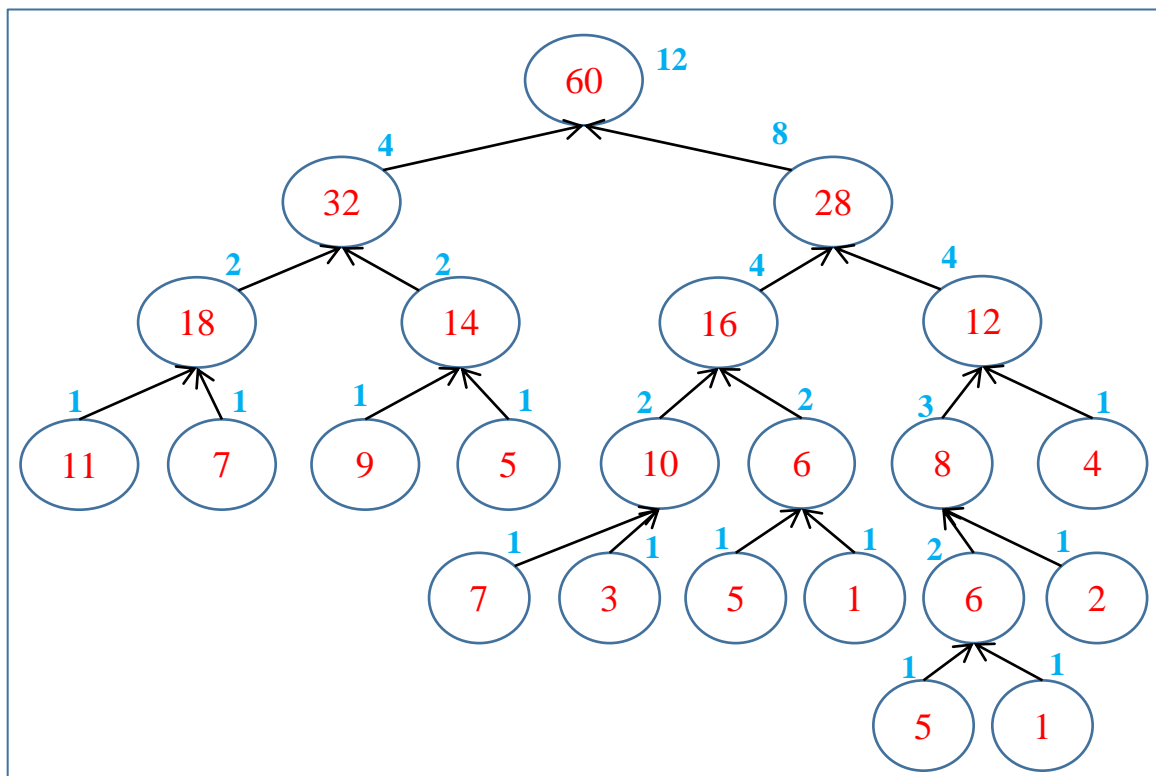
Một nhóm có  $n$  người thì có thể tách thành 2 nhóm với số thành viên lớn hơn không và chênh nhau đúng 4 người. Chẳng hạn một nhóm có  $n = 31$  người thì không thể tách được thành 2 nhóm nào, còn  $n = 60$  người thì tách thành 2 nhóm có 32 và 28 người; hai nhóm này lại tiếp tục tách tiếp tới khi không thể tách được nữa thì dừng lại. Bài toán đặt ra là cứ như vậy tách tới khi các nhóm không thể tách được nữa thì có bao nhiêu nhóm?

Hình vẽ dưới đây thể hiện cấu trúc cây mô tả cách tách nhóm được thực hiện từ trên xuống và tổng hợp số liệu từ dưới lên với nhóm ban đầu có số người là 60 thực hiện tách tất cả ra thành 12 nhóm.

## Chia



## Trị và liên kết



Phương pháp chia để trị ở đây muốn biết  $n$  người được tách thành bao nhiêu nhóm? Ta thấy nếu  $n$  là số chẵn thì tách thành 2 nhóm một nhóm  $n/2+2$  người và một nhóm  $n/2-2$  người, đây là hai bài toán con có cùng cách giải chúng ta chỉ việc dùng đệ quy giải hai bài toán này và tổng lại là xong. Ngược lại với những bài toán con suy biến khi số người  $n$  là số lẻ hoặc ít hơn hoặc bằng 4 thì không tách nữa mà chỉ có một nhóm.

### Chương trình

```
#include<stdio.h>
int tach(int n)
{
    if(n<=4 || n%2) return 1;
    return tach(n/2+2)+tach(n/2-2);
}
int main()
{
    int n;
    printf("Nhap vao so nguoi : ");    scanf("%d",&n);
    printf("So nhom can tim %d",tach(n));
}
```

Độ phức tạp thuật toán: Đặt  $T(n)$  là độ phức tạp thuật toán trong trường hợp tốt nhất không tách được nhóm dừng luôn  $T(n) = \Omega(1)$ , còn trường hợp xấu nhất mất 2 lần gọi đệ quy mỗi lần xấp xỉ  $n/2$  do đó  $T(n) = 2T(n/2) + c$  áp dụng định lý Master thu gọn ta có  $T(n) = O(n)$ .

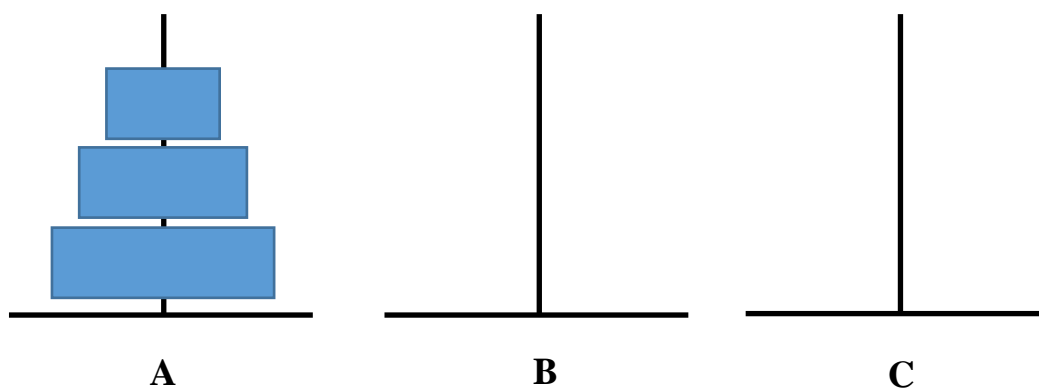
Kỹ thuật chia để trị là kỹ thuật áp dụng giải quyết được nhiều thuật toán hiệu quả, chẳng hạn như thuật toán sắp xếp (sắp xếp nhanh, sắp xếp trộn), thuật toán nhân (thuật toán Karatsuba nhân 2 số, thuật toán Strassen nhân ma trận), thuật toán phân tích cú pháp trong xử lý ngôn ngữ tự nhiên, thuật toán biến đổi Fourier rời rạc.

Tư duy để thiết kế thuật toán chia để trị đòi hỏi khả năng khái quát hóa, trừu tượng hóa bài toán, các mô hình xây dựng theo quy nạp toán học hoặc có biểu thức truy hồi tính toán trong đó thể hiện mối quan hệ đệ quy giữa các

bước tính toán. Mỗi bài toán chia để trị lại có một đặc thù riêng và không có một phương pháp có hệ thống nào để tìm ra bài toán tổng quát thích hợp trong mọi trường hợp.

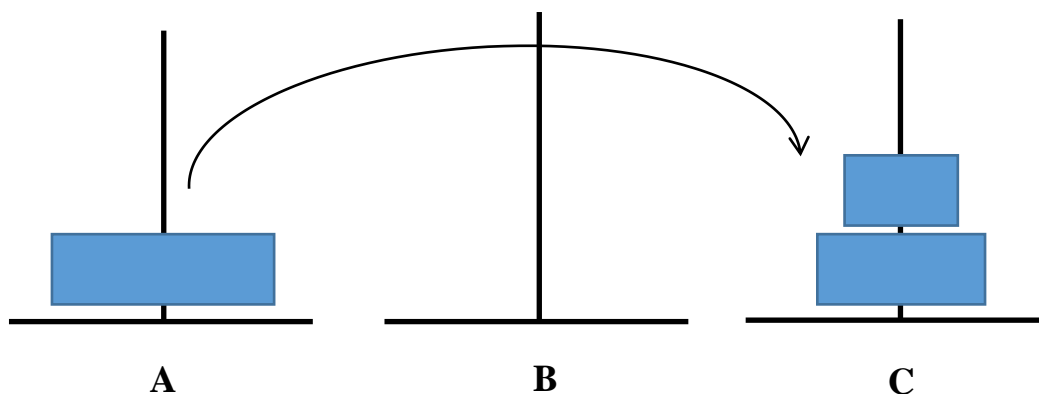
### ***Ví dụ 2: Bài toán tháp Hà Nội***

Bài toán cho  $n$  cái đĩa kích thước đôi một khác nhau có lỗ ở giữa để có thể lồng vào ba cái trục gồm: A là trục nguồn, B là trục đích, và C là trục trung chuyển. Đầu tiên, những cái đĩa này được xếp tại trục A. Vậy làm thế nào để chuyển toàn bộ các đĩa sang trục B, với điều kiện mỗi lần chuyển chuyển từng cái một và luôn phải đảm bảo quy định "nhỏ trên lớn dưới", biết rằng trục C được phép sử dụng làm trục trung chuyển.



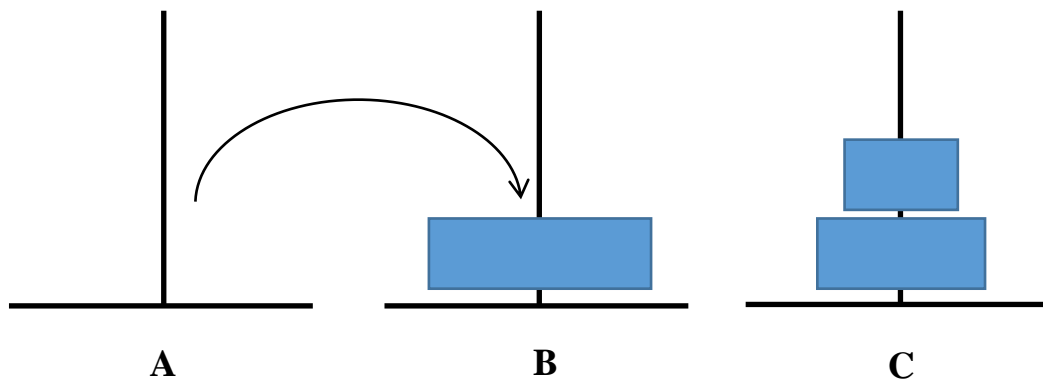
Chúng ta thấy rằng nếu chỉ có một đĩa thì ta sẽ di chuyển trực tiếp từ trục A sang trục B. Ngược lại ta cần di chuyển  $n - 1$  đĩa từ trục A sang trục C sử dụng trục B làm trung gian sau đó di chuyển đĩa cuối cùng to nhất từ trục A sang trục B và di chuyển  $n - 1$  đĩa từ trục C sang trục B sử dụng trục A làm trung gian theo ba bước sau:

#### **Bước 1:**

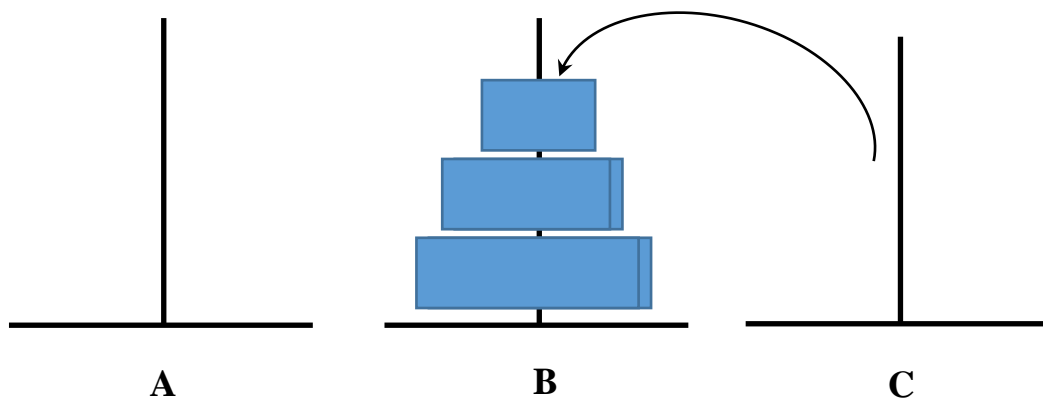




**Bước 2:**



**Bước 3:**



Trên cơ sở cách phân tích như vậy chúng ta có chương trình như sau:

**Chương trình:**

```
#include<iostream>
void ThapHN(int n,char A,char B,char C)
{
    if(n>1) ThapHN(n-1,A,C,B);
    std::cout<<"\nChuyen dia "<<n<<" tu "<<A<<" sang "<<B;
    if(n>1) ThapHN(n-1,C,B,A);
}
int main()
{
    int n;
    std::cout<<"Nhap vao so dia n : "; std::cin>>n;
    ThapHN(n,'A','B','C');
}
```

Đánh giá độ phức tạp của thuật toán: Đặt  $T(n)$  là độ phức tạp của thuật toán ta có  $T(n) = 2T(n-1) + c = 4T(n-2) + 2c = \dots = 2^n T(n-n) + nc$  hay  $T(n) = 2^n T(0) + nc = c_0 2^n + nc = \Theta(2^n)$ .

Trong trường hợp muốn tính số phép chuyển đĩa mà không cần quan tâm đến cách chuyển tháp Hà Nội, ta thấy rằng số phép chuyển đĩa ở bước 1 (chuyển  $n-1$  đĩa từ trục A sang trục C) bằng đúng số phép chuyển đĩa ở bước 3 (chuyển  $n-1$  đĩa từ trục C sang trục B) do đó thuật toán ta chỉ cần gọi một lần đệ quy với chương trình như sau:

### **Chương trình**

```
#include<stdio.h>
long ThapHN(int n)
{
    return n==1?1:2*ThapHN(n-1)+1;
}
int main()
{
    int n;
    printf("Nhap vao so dia n : ");    scanf("%d",&n);
    printf("So buoc chuyen dia %ld",ThapHN(n));
}
```

Đoạn chương trình này có độ phức tạp  $T(n) = T(n-1) + c = \dots = \Theta(n)$ . Thực ra bằng toán rời rạc dễ dàng chứng minh được số phép chuyển tháp là  $2^n - 1$  để tính giá trị này ta dùng phép dịch  $n$  bit là xong.

### **3.2. Một số thuật toán giảm đệ trị**

Kỹ thuật chia để trị đôi khi cũng được áp dụng cho các thuật toán quy bài toán ban đầu về đúng một bài toán nhỏ hơn, những thuật toán này có thể được lập trình hiệu quả hơn thuật toán chia để trị thông thường và nhiều khi các thuật toán này được khử đệ quy bằng cách chuyển về đệ quy đuôi rồi chuyển vòng lặp hoặc sử dụng cấu trúc dữ liệu ngăn xếp. Những thuật toán như vậy đôi khi được gọi là giảm đệ trị.

**Ví dụ: Tính giai thừa của  $n$  bằng đệ quy đuôi**

Thuật toán đệ quy đuôi là thuật toán đệ quy và có lời gọi đệ quy ở cuối cùng sau khi đã tính toán hết, khi thực hiện đệ quy đuôi thì các chương trình dịch sẽ không phải sử dụng cấu trúc dữ liệu ngăn xếp để lưu vết và truy vết tìm ra kết quả mà chạy thẳng như sử dụng vòng lặp.

Để tính giai thừa chúng ta sử dụng một tham biến  $k$  khởi gán bằng 1 chứa kết quả đầu ra của  $n!$  và thực hiện đệ quy đuôi với bản chất là nhân ngược từ  $n$  về 1 thu được giá trị  $n!$ .

**Chương trình tính  $n!$  bằng thuật toán đệ quy đuôi, giảm để trị**

```
#include<stdio.h>
long long giaithua(int n,long long &k)
{
    if(n==0||n==1) return k;
    k=k*n;
    return giaithua(n-1,k);
}
int main()
{
    int n;
    long long k=1;
    printf("Nhap vao so n : ");    scanf("%d",&n);
    printf("Gia tri giai thua la %lld",giaithua(n,k));
}
```

Đánh giá độ phức tạp của thuật toán: Đặt  $T(n)$  là độ phức tạp của thuật toán ta có  $T(n) = T(n-1) + c = T(n-2) + 2c = \dots = T(n-n) + nc = T(0) + nc$  hay độ phức tạp thuật toán là  $T(n) = \Theta(n)$ .

**3.2.1. Bài toán tìm ước chung lớn nhất của hai số**

Bài toán cho hai số nguyên  $a$  và  $b$  tìm ước chung lớn nhất.

Theo định lý Euclid, nếu gọi  $a \bmod b$  là phần dư của phép chia  $a$  cho  $b$  thì ước chung lớn nhất của hai số  $a$  và  $b$  là  $\gcd(a,b)$  được phát biểu dưới dạng biểu thức truy hồi như sau:

$$\gcd(a,b) = \begin{cases} |a| & \text{Khi } b = 0 \\ \gcd(b, a \bmod b) & \text{Khi } b \neq 0 \end{cases}$$

Do đó ta có thuật toán giảm để trị thay vì tìm ước chung lớn nhất của  $a$  và  $b$  ta gọi đệ quy tìm ước chung lớn nhất của  $b$  và  $a \bmod b$  tới khi suy biến  $b = 0$  thì dừng ta thu được  $a$  là ước chung lớn nhất của hai số.

```
long long gcd(long long a, long long b)
{
    return b ? gcd(b, a % b) : (a > 0 ? a : -a);
}
```

Độ phức tạp của thuật toán trong trường hợp tốt nhất vừa vào hoặc sau một số bước thì  $b = 0$ , nên độ phức tạp tốt nhất là  $\Omega(1)$ , trong trường hợp xấu nhất ước chung lớn của hai số Fibonacci liên tiếp biến đổi từ từ về 1. Chẳng hạn ước chung lớn nhất của 89 và 55 được biến đổi như sau:

$$\begin{aligned} \gcd(89, 55) &= \gcd(55, 34) = \gcd(34, 21) = \gcd(21, 13) \\ &= \gcd(13, 8) = \gcd(8, 5) = \gcd(5, 3) \\ &= \gcd(3, 2) = \gcd(2, 1) = \gcd(1, 0) = 1. \end{aligned}$$

Chúng ta thấy rằng tốc độ tăng trưởng của dãy Fibonacci là một dãy đơn điệu tăng thỏa mãn  $F_n = F_{n-2} + F_{n-1}$  do  $0 < F_{n-2} < F_{n-1}$  suy ra

$2F_{n-2} < F_n < 2F_{n-1}$  nên độ phức tạp sẽ là:

$$T(a,b) = \max(O(\log(a)), O(\log(b))).$$

### 3.2.2. Bài toán xóa chữ số

**Bài toán:** Cho một số có nguyên dương  $n$  chữ số, và một số nguyên dương  $k$  sao cho  $1 \leq k < n \leq 10^5$ . Nhiệm vụ của bạn là xóa đi đúng  $k$  chữ số của số có  $n$  chữ số đó để được số còn lại giữ nguyên thứ tự sau khi xóa là lớn nhất có thể. Chẳng hạn số 4728168 có  $n = 7$  chữ số thì:

- Với  $k = 1$ , ta xóa đi chữ số 4 được số lớn nhất là 728168.

- Với  $k = 2$ , ta xóa đi các chữ số 4, 2 được số lớn nhất là 78168.
- Với  $k = 3$ , ta xóa đi chữ số 4, 7, 2 được số lớn nhất là 8168.
- Với  $k = 4$ , ta xóa đi chữ số 4, 7, 2, 1 được số lớn nhất là 868.
- Với  $k = 5$ , ta xóa đi chữ số 4, 7, 2, 1, 6 được số lớn nhất là 88.
- Với  $k = 6$ , ta xóa đi chữ số 4, 7, 2, 8, 1, 6 được số lớn nhất là 8.

**Phân tích thuật toán giảm để trị:** Ta xét số nguyên dương dưới dạng một xâu các chữ số có độ dài  $n$ , ta muốn xóa đi  $k$  chữ số để được số lớn nhất

**Trường hợp bài toán suy biến:**  $k = 0$  không phải xóa gì cả, nếu  $k$  bằng độ dài xâu bằng  $n$  thì xóa tất cả.

**Trường hợp bài toán chưa suy biến:**  $k > 0$  ta tìm vị trí giá trị lớn nhất đầu tiên trong  $k+1$  chữ số đầu tiên ở vị trí  $m$  khi đó ta coi như xóa đi  $m$  chữ số đầu giữ lại chữ số thứ  $m$ , sau đó gọi đệ quy giảm để trị xóa đi  $k - m$  chữ số của số bắt đầu từ  $m+1$ .

**Minh họa:** Số 4728168 có  $n = 7$  muốn xóa đi  $k = 5$  chữ số với hàm `Xoa(4728168, 3)` được thực hiện như sau:

**Bước 1.** Tìm  $\max(4, 7, 2, 8, 1, 6) = 8$  ở vị trí 4 nên xóa đi 3 chữ số đầu đưa bài toán về giảm để trị còn `Xoa(8168, 2)`.

**Bước 2.** Tìm  $\max(8, 1, 6) = 8$  ở vị trí đầu nên xuất ra chữ số '8' và đưa bài toán về giảm để trị còn `Xoa(168, 2)`.

**Bước 3.** Tìm  $\max(1, 6, 8) = 8$  ở vị trí thứ 3 nên xóa đi 2 chữ số đầu đưa bài toán về giảm để trị `Xoa(8, 0)` trường hợp này suy biến xuất nốt '8'.

### Chương trình

```
#include <stdio.h>
#include <string.h>
void xoa(int k, char *x)
{
    if(k==strlen(x)) return;
    if(k==0) {printf("%s",x);return;}
    int p=0;
    for(int i=1;i<=k;i++)
        if(x[i]>x[p])p=i;
```

```
    printf("%c",x[p]);  
    xoa(k-p,x+p+1);  
}  
int main()  
{  
    int k;  
    char x[100005];  
    printf("Nhap vao so ban dau : ");  
    scanf("%s",x);  
    printf("Nhap so k = "); scanf("%d",&k);  
    xoa(k,x);  
}
```

Đánh giá độ phức tạp thuật toán: Đặt  $T(n,k)$  là độ phức tạp thuật toán. Trong trường hợp tốt nhất thì duyệt  $k$  bước gọi đệ quy chỉ còn xóa 0 phần tử nên  $T(n,k) = \Omega(k)$ . Trong trường hợp xấu nhất mỗi lần duyệt  $k$  bước nhưng tìm ra vị trí lớn nhất lại ở đầu tiên khi đó độ phức tạp  $T(n,k) = O(n * k)$ .

### 3.2.3. Bài toán tính lũy thừa

**Bài toán:** Cho một số có nguyên không âm  $n$  và một số thực  $x$ , hãy viết chương trình tính lũy thừa  $x^n$

**Phân tích thuật toán giảm để trị:** Với số thực  $x$  bất kỳ và số nguyên không âm  $n$ , chúng ta có biểu thức truy hồi

$$x^n = \begin{cases} 1 & \text{Khi } n = 0 \\ (x^{n/2})^2 & \text{Khi } n > 0, n \bmod 2 = 0 \\ x \times (x^{\lfloor n/2 \rfloor})^2 & \text{Khi } n > 0, n \bmod 2 = 1 \end{cases}$$

Do đó để tính  $x^n$  ta làm như sau:

**Trường hợp bài toán suy biến:**  $n = 0$  thì giá trị cần tính là 1

**Trường hợp bài toán chưa suy biến:**  $n > 0$ , chúng ta đưa về đệ quy bài toán nhỏ hơn tính  $x^{\lfloor n/2 \rfloor}$  sau đó bình phương lên nếu  $n$  lẻ thì nhân thêm một  $x$ .

**Chương trình:**

```
#include<stdio.h>  
double luythua(double x,long n)  
{
```

```
    if(n==0) return 1.0;
    double t=luythua(x,n/2);
    t=t*t;
    return n%2?t*x:t;
}
int main(){
    double x;
    long n;
    printf("Nhap x = "); scanf("%lf",&x);
    printf("Nhap n = "); scanf("%ld",&n);
    printf("Gia tri x luy thua n la : %lf",luythua(x,n));
}
```

Đánh giá độ phức tạp của thuật toán: Chúng ta đặt  $T(n)$  là độ phức tạp của thuật toán, theo chương trình ta có biểu thức truy hồi

$$T(n) = \begin{cases} c_o & \text{Khi } n = 1 \\ T(n/2) + c & \text{Khi } n > 1 \end{cases}$$

theo dụng định lý Master thu gọn ta có  $T(n) = \Theta(\log n)$ .

### 3.2.4. Bài toán tìm số Fibonacci thứ $n$

**Bài toán:** Cho số nguyên dương  $n$  tính số Fibonacci thứ  $n$  theo công thức:

$$F_n = \begin{cases} 1 & \text{khi } n = 1, 2 \\ F_{n-1} + F_{n-2} & \text{khi } n \geq 3 \end{cases}$$

**Phân tích thuật toán giảm để trị:** Chúng ta sử dụng kết quả toán học (đã chứng minh bằng quy nạp toán): Với mọi số tự nhiên  $n \geq 2$  và các  $F_{n-1}, F_n, F_{n+1}$

là các số Fibonacci thứ  $n-1, n, n+1$  ta đều có  $\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$

Do đó bài toán đưa về bài toán tìm lũy thừa thứ  $n$  của ma trận cấp 2, chúng ta áp dụng thuật toán giảm để trị giống bài lũy thừa số thực ở mục trên có điều thay vì số thực thì ta dùng ma trận. Hơn nữa theo tính chất của dãy Fibonacci ta thấy rằng lũy thừa ma trận là ma trận cấp 2 đối xứng có dạng:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} a & b \\ b & a-b \end{pmatrix}$$

Sau khi tính xong thì số Fibonacci thứ  $n$  chính là  $b$ .

**Trường hợp bài toán suy biến:**  $n = 1$  thì  $a = 1$  và  $b = 1$

**Trường hợp bài toán chưa suy biến:**  $n > 1$ , chúng ta đưa về đệ quy bài toán

nhỏ hơn tính  $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{\lfloor n/2 \rfloor} = \begin{pmatrix} x & y \\ x & x-y \end{pmatrix}$ , tùy theo  $n$  chẵn hay lẻ ta tính  $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$

với hai trường hợp:

Khi  $n$  chẵn thì:

$$\begin{pmatrix} a & b \\ b & a-b \end{pmatrix} = \begin{pmatrix} x & y \\ y & x-y \end{pmatrix} \times \begin{pmatrix} x & y \\ y & x-y \end{pmatrix} = \begin{pmatrix} x^2 + y^2 & 2xy - y^2 \\ 2xy - y^2 & x^2 - 2xy + 2y^2 \end{pmatrix}$$

Khi  $n$  lẻ thì

$$\begin{aligned} \begin{pmatrix} a & b \\ b & a-b \end{pmatrix} &= \begin{pmatrix} x & y \\ y & x-y \end{pmatrix} \times \begin{pmatrix} x & y \\ y & x-y \end{pmatrix} \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \\ &= \begin{pmatrix} x^2 + y^2 & 2xy - y^2 \\ 2xy - y^2 & x^2 - 2xy + 2y^2 \end{pmatrix} \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \\ &= \begin{pmatrix} x^2 + 2xy & x^2 + y^2 \\ x^2 + y^2 & 2xy - y^2 \end{pmatrix} \end{aligned}$$

**Chương trình:**

```
#include <stdio.h>
void LuyThuaMaTran(int n, long &a, long &b)
{
    if(n == 1) {a=b=1; return;}
    long x,y;
    LuyThuaMaTran(n/2,x,y);
    if(n%2==0)
    {
        a=x*x+y*y;
        b=2*x*y-y*y;
    }
    else
    {
        a=x*x+2*x*y;
        b=x*x+y*y;
    }
}
```



```
    }  
}  
int main()  
{  
    int n;  
    long u,v;  
    printf("Nhap so nguyen duong n = "); scanf("%d",&n);  
    LuyThuaMaTran(n,u,v);  
    printf("F[%d]=%ld\n",n,v);  
}
```

Độ phức tạp thuật toán tương tự như bài lũy thừa cũng có độ phức tạp là  $T(n) = \Theta(\log n)$ .

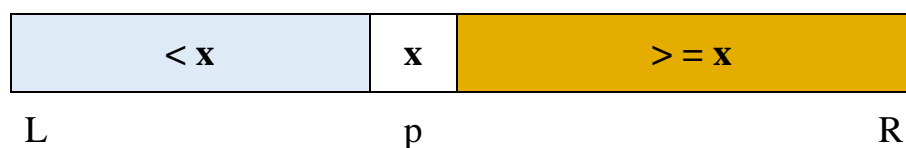
### 3.2.5. Bài toán tìm số lớn thứ k trong dãy n phần tử

**Bài toán:** Cho dãy số nguyên  $a_1, a_2, \dots, a_n$  có n phần tử và một số nguyên dương k ( $1 \leq k \leq n$ ). Bài toán đặt ra nếu ta sắp xếp tăng dần thì phần tử thứ k của dãy là số nào? Chẳng hạn dãy {4, 8, 2, 7, 9, 3, 5} thì phần tử thứ 3 của dãy là số 4, còn phần tử thứ 6 của dãy là 8.

**Phân tích thuật toán giảm để trị:** Chúng ta làm tương tự phân hoạch của thuật toán sắp xếp nhanh Quicksort. Trước hết, chúng ta tổng quát hóa bài toán tìm phần tử thứ k của dãy  $a_L, a_{L+1}, \dots, a_R$  từ chỉ số L tới chỉ số R.

**Trường hợp bài toán suy biến:** Dãy có một phần tử tức là  $L = R$  thì  $k = 1$  phần tử cần tìm là  $a[L]$ .

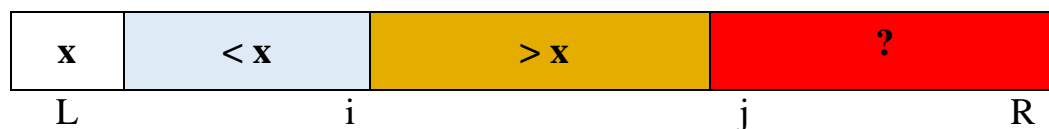
**Trường hợp bài toán chưa suy biến:** Dãy có nhiều hơn một phần tử tức là  $L < R$ , ta chọn một phần tử chốt x bất kỳ trong dãy và phân hoạch dãy thành 2 phần dựa vào phần tử chốt x sao cho những phần tử nhỏ hơn x nằm sang bên trái của x và phần tử lớn hơn x sẽ nằm sang bên phải của x.



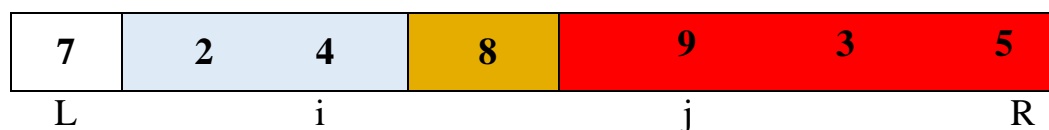
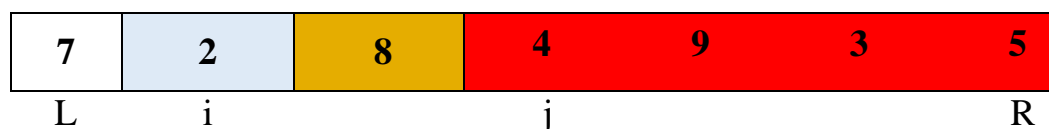
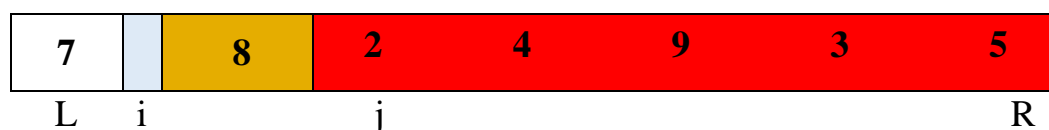
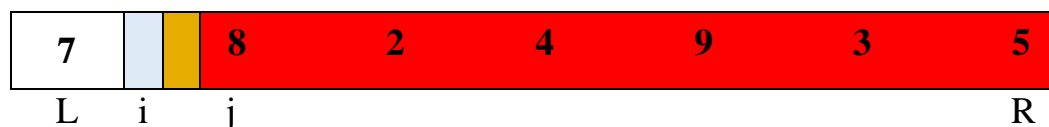
Ở đây nếu sắp xếp dãy thì x đã đứng đúng vị trí, giả sử vị trí của x là p ta có:

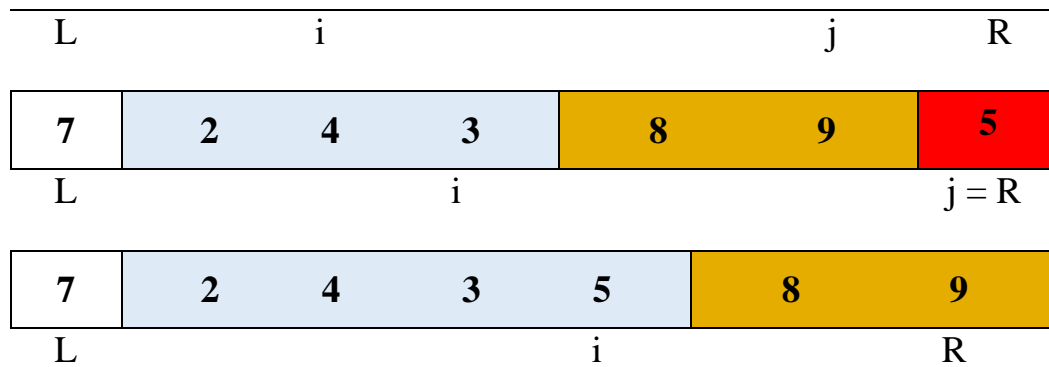
- Nếu  $p - L + 1 = k$  thì  $x$  là phần tử thứ  $k$  cần tìm
- Nếu  $p - L + 1 > k$  thì phần tử cần tìm ở bên trái của  $x$  ta gọi đệ quy giảm để tìm phần tử thứ  $k$  ở nửa bên trái từ  $L$  đến  $p - 1$
- Nếu  $p - L + 1 < k$  thì phần tử cần tìm ở bên phải của  $x$  ta gọi đệ quy giảm để tìm phần tử thứ  $k - (p - L + 1)$  ở nửa bên phải từ  $p + 1$  tới  $R$ .

*Vấn đề còn lại là phân chọn  $x$  và phân hoạch:* Cách làm giống như phân hoạch của thuật toán sắp xếp nhanh Quicksort, nhưng cũng có nhiều cách. Ở đây chúng ta lấy phần tử ở giữa làm phần tử chốt  $x = a[(L+R)/2]$  sau đó hoán đổi với  $a[L]$  để thuận lợi xử lý. Ví dụ có 7 phần tử  $\{4, 8, 2, 7, 9, 3, 5\}$  từ  $a[1]$  đến  $a[7]$  thì phần tử chốt là  $a[(1+7)/2] = a[4] = 7$  là  $x$  đổi chỗ với  $a[1]$  ta được dãy  $\{7, 8, 2, 4, 9, 3, 5\}$ . Chúng ta xuất phát tập nhỏ hơn  $a[1]$  và lớn hơn  $a[1]$  ban đầu đều rỗng rồi xét lần lượt các phần tử  $\{8, 2, 4, 9, 3, 5\}$  cứ nhỏ hơn thì đưa vào phần nhỏ hơn, lớn hơn đưa vào phần lớn hơn

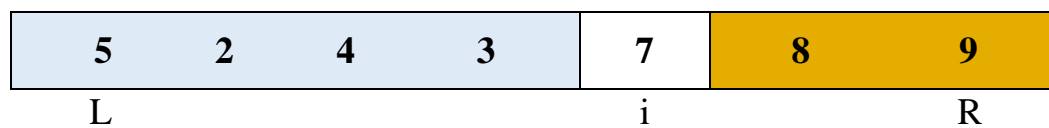


Cụ thể, nếu xét  $a[j] > a[L]$  thì ta bổ sung vào phần lớn hơn nên không làm gì; ngược lại  $a[j] < a[L]$  ta phải bổ sung  $a[j]$  vào phần nhỏ hơn bằng cách tăng  $i$  lên 1 đơn vị và đổi chỗ  $a[i]$ ,  $a[j]$  với nhau theo minh họa:





Bước cuối cùng ta đổi chỗ  $a[L]$  với  $a[i]$  để đưa phần tử chốt vào giữa dãy ở vị trí  $i$  ta có phân hoạch:



### Chương trình

```
#include<stdio.h>
void Swap(int &a,int &b)
{
    int t=a;a=b;b=t;
}
void Nhap(int &n,int *&a)
{
    printf("Nhap so phan tu n = "); scanf("%d",&n);
    a=new int [n+1];
    for(int i=1;i<=n;i++)
    {
        printf("a[%d] = ",i);
        scanf("%d",&a[i]);
    }
}
int PhanHoach(int *a,int L,int R)
{
    Swap(a[L],a[(L+R)/2]);
    int i=L;
    for(int j=L+1;j<=R;j++)
        if(a[L]>a[j]) Swap(a[++i],a[j]);
    Swap(a[L],a[i]);
    return i;
}
int TimPhanTuThu_k(int *a,int k,int L,int R)
```

```
{
    if(L==R) return a[L];
    int p=PhanHoach(a,L,R);
    if(p-L+1==k) return a[p];
    if(p-L+1>k) return TimPhanTuThu_k(a,k,L,p-1);
    return TimPhanTuThu_k(a,k-(p-L+1),p+1,R);
}
int main()
{
    int *a,n,k;
    Nhap(n,a);
    printf("Nhap k = "); scanf("%d",&k);
    printf("Phantu thu %d la
%d",k,TimPhanTuThu_k(a,k,1,n));
    delete []a;
}
```

Đánh giá độ phức tạp của thuật toán: Đối với dãy có  $n$  phần tử ta đặt độ phức tạp  $T(n)$ , do độ phức tạp phân hoạch gồm một vòng lặp là  $cn$  ta có:

+ Trường hợp tốt nhất vừa phân hoạch ta tìm ra kết quả thì độ phức tạp của thuật toán  $T(n) = \Omega(n)$ .

+ Trường hợp xấu nhất mỗi lần phân hoạch không đều 2 bên mà bên có bên không thì

$$\begin{aligned} T(n) &= T(n-1) + cn = T(n-2) + c(n-1) + cn = \dots \\ &= T(1) + c \times 2 + c \times 3 + \dots + cn = O(n^2) \end{aligned}$$

+ Trường hợp trung bình phân hoạch tương đối đều hai bên ta có  $T(n) = T(n/2) + cn$ , theo trường hợp 3 định lý Master thu gọn ta có  $T(n) = O(n)$ .

### **3.2.6. Bài toán Robot cắt thanh kim loại**

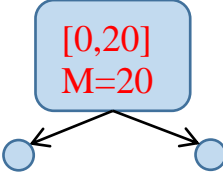
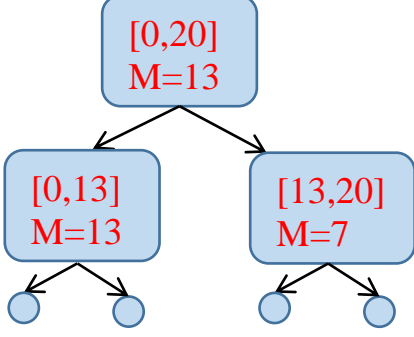
**Bài toán:** Một con Robot được thiết kế để cắt thanh kim loại nằm ngang ban đầu có độ dài  $m$  đơn vị độ dài ( $1 < m < 10^6$ ), thành các đoạn nhỏ. Robot được lập trình  $n$  ( $1 < n < 10^5$ ) lần cắt mỗi lần các mép trái thanh lần lượt là  $a_1, a_2, \dots, a_n$  đơn vị độ dài, sao cho hai lần cắt không bao giờ cùng một vị trí. Người sử dụng Robot muốn biết mỗi lần cắt xong thì độ dài dài nhất của các

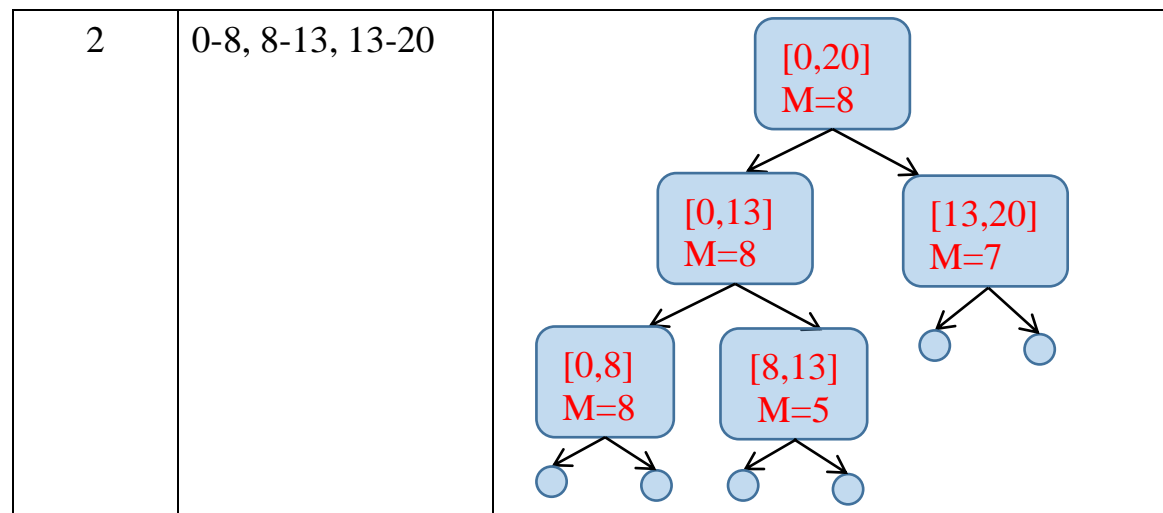
đoạn con là bao nhiêu. Bạn hãy lập trình nhập vào hai số nguyên dương  $m, n$  và dãy số nguyên dương  $a_1, a_2, \dots, a_n$ , chỉ ra  $n$  giá trị mỗi giá trị là độ dài đoạn con thu được dài nhất cần tìm.

**Chẳng hạn:** Cho thanh kim loại dài  $m = 20$  (đơn vị độ dài) và 5 lần cắt lần lượt cách mép trái của thanh ban đầu là 13, 8, 17, 5, 9 thì kết quả minh họa trong bảng:

Bước	Cắt tại	Độ dài các thanh	Xuất ra
Khởi tạo		0-20	
1	13	0-13, 13-20	$\max(13, 7) = 13$
2	08	0-8, 8-13, 13-20	$\max(8, 5, 7) = 8$
3	17	0-8, 8-13, 13-17, 17-20	$\max(8, 5, 4, 3) = 8$
4	05	0-5, 5-8, 8-13, 13-17, 17-20	$\max(5, 3, 5, 4, 3) = 5$
5	09	0-5, 5-8, 8-9, 9-13, 13-17, 17-20	$\max(5, 3, 1, 4, 4, 3) = 5$

**Phân tích thuật toán giảm để trị:** Chúng ta xây dựng cấu trúc dữ liệu cây nhị phân, mỗi nút quản lý các đoạn cắt gồm vị trí nút trái và phải và độ dài đoạn con dài nhất của nó theo mô hình sau:

Bước	Độ dài các thanh	Cây
Khởi tạo	0-20	
1	0-13, 13-20	



**Trường hợp bài toán suy biến:** Cây chỉ có nút gốc  $[a, b, M]$  trong đó  $a$  là nút trái,  $b$  là nút phải,  $M$  là đoạn con dài nhất chính là  $b-a$ . Chúng ta cắt tại vị trí  $v$  nào đó thì sẽ thêm 2 cây gồm: cây con trái  $[a, v, v-a]$  không có con và cây con phải  $[v, b, b-v]$  không có con.

**Trường hợp bài toán chưa suy biến:** Nút gốc  $[a, b, M]$  chắc chắn có hai cây con, nếu vết cắt tại  $v < a$  của cây con phải thì cắt vào cây con trái, chúng ta gọi đệ quy giảm đệ trị vào cây con trái, ngược lại  $v > a$  của cây con phải thì cắt vào cây con phải, chúng ta gọi đệ quy vào cây con phải, sau đó cập nhật lại giá trị đoạn con dài nhất của gốc dựa vào đoạn con dài nhất của 2 cây con trái và phải.

**Chương trình:**

```

#include<stdio.h>
#define Max(a,b) (a)>(b)?(a):(b)
typedef struct Tree
{
    long a,b,M;
    Tree *Left,*Right;
    Tree(long trai=0,long phai=0) //ham tao
    {
        a=trai; b=phai;
        M=b-a;
        Left=0;Right=0;
    }
}Tree;
void Add(Tree *T,long v)
    
```

```
{
    if(T->Left==0 && T->Right==0) //trường hợp suy biến
    {
        T->Left=new Tree(T->a,v);
        T->Right=new Tree(v,T->b);
    }
    else if(T->Left->b>v) Add(T->Left,v);
    else Add(T->Right,v);
    T->M=Max(T->Left->M,T->Right->M);
}
int main()
{
    long m,n,v;
    scanf("%ld%ld",&m,&n);
    Tree *T=new Tree(0,m);
    long *KQ=new long [n+5];
    for(long i=1;i<=n;i++)
    {
        scanf("%ld",&v);
        Add(T,v);
        KQ[i]=T->M;
    }
    printf("\nKet qua cac doan con dai nhat\n");
    for(long i=1;i<=n;i++) printf("%ld\n",KQ[i]);
}
```

**Đánh giá độ phức tạp:** Đặt độ phức tạp thuật toán là  $T(n)$ , chúng ta xét một thao tác cắt khi ở lần cắt thứ  $n + 1$  khi đã cắt  $n$  lần. Trong trường hợp tốt nhất lúc cây nhị phân cũng cân bằng có  $n$  nút thao tác cập nhật sẽ tốn  $\log n$  thao tác để duyệt xuống lá và cập nhật trở lại gốc nên tổng từ đầu đến  $n$  bước sẽ là  $T(n) = \Omega(n \log n)$ . Trong trường hợp xấu nhất cây nhị phân trở thành một cây tuyến tính thẳng tuột không rẽ nhánh sẽ tốn  $n$  thao tác để duyệt xuống lá và cập nhật lại gốc nên tổng từ đầu đến  $n$  bước thì độ phức tạp  $T(n) = O(n^2)$ .

### **3.3. Một số thuật toán chia để trị**

#### **3.3.1. Bài toán leo thang**

**Bài toán:** Một chiếc cầu thang có  $n$  bậc, một người nào đó có thể bước một bước lên một bậc, hai bậc hoặc ba bậc. Hỏi để lên hết  $n$  bậc thang thì người đó có bao nhiêu cách leo thang.

**Phân tích thuật toán:** Đặt  $L_n$  là số cách leo lên  $n$  bậc cầu thang ta có

- Nếu  $n = 1$  thì có 1 cách lên 1 bậc nên  $L_1 = 1$ .
- Nếu  $n = 2$  thì có 2 cách lên 2 bậc gồm bước từng bước mỗi bước một bậc hoặc là bước một bước luôn hai bậc nên  $L_2 = 2$ .
- Nếu  $n = 3$  thì có 4 cách lên 3 bậc gồm: bước từng bước mỗi bước một bậc hoặc bước hai bậc rồi một bậc hoặc bước một bậc rồi hai bậc hoặc là bước một bước luôn ba bậc nên  $L_3 = 4$ .
- Nếu  $n \geq 3$  thì để lên được bậc  $n$  thì có đúng 3 cách lên hoặc là đứng ở bậc thứ  $n - 3$  bước một bước 3 bậc hoặc là đứng ở bậc thứ  $n - 2$  bước một bước 2 bậc hoặc là đứng ở bậc thứ  $n - 1$  bước một bước 1 bậc nên ta có biểu thức  $L_n = L_{n-3} + L_{n-2} + L_{n-1}$ .

**Thiết kế thuật toán:**

**Trường hợp bài toán suy biến thứ nhất:** Chỉ có một bậc thì  $L_1 = 1$ .

**Trường hợp bài toán suy biến thứ hai:** Chỉ có hai bậc thì  $L_2 = 2$ .

**Trường hợp bài toán suy biến thứ hai:** Chỉ có ba bậc thì  $L_3 = 4$ .

**Trường hợp bài toán chưa suy biến:** Khi số bậc  $n \geq 3$

- + **Chia:** Chia bài toán ban đầu thành ba bài toán con với  $n - 3$ ,  $n - 2$  và  $n - 1$  bậc.
- + **Trị:** Gọi đệ quy tính kết quả các bài toán con này
- + **Liên kết:** Dem tính tổng kết quả của ba bài toán con được kết quả bài toán ban đầu.

**Chương trình**



```
#include<stdio.h>
long LeoThang(int n)
{
    if(n==1) return 1;
    if(n==2) return 2;
    if(n==3) return 4;
    return LeoThang(n-3)+LeoThang(n-2)+LeoThang(n-1);
}
int main()
{
    int n;
    printf("Nhap vao so bac n : "); scanf("%d",&n);
    printf("So cach leo thang la %ld",LeoThang(n));
}
```

**Đánh giá độ phức tạp của thuật toán:** Đặt  $T(n)$  là độ phức tạp của thuật toán

$$\text{ta có } T(n) = \begin{cases} c_0 & \text{Khi } n < 3 \\ T(n-1) + T(n-2) + T(n-3) + c & \text{Khi } n \geq 3 \end{cases}$$

Dễ thấy  $T(n)$  là dãy đơn điệu tăng nên  $3T(n-3) < T(n) < 3T(n-1)$  suy ra  $3^h T(n-3h) < T(n) < 3^k T(n-k)$ . Nếu ta chọn  $h$  sao cho  $n-3h < 3$  thì  $h \approx n/3$  và  $T(n-3h) = c_0$ ; chọn  $k$  sao cho  $n-k < 3$  thì  $k \approx n$  và  $T(n-k) = c_0$  khi đó  $3^{n/3} c_0 < T(n) < 3^n c_0$ . Vậy  $T(n) = \Omega(3^{n/3})$  và  $T(n) = O(3^n)$ .

**Cải tiến thuật toán thông qua sử dụng bộ nhớ trung gian:** Như đánh giá ở trên thì độ phức tạp của thuật toán là hàm mũ, để tính  $L_n$  ta phải tính  $L_{n-1}, L_{n-2}, L_{n-3}$  nhưng để tính  $L_{n-1}$  trong đó lại có  $L_{n-2}, L_{n-3}$  nó không chỉ tính đi tính lại  $L_{n-2}, L_{n-3}, L_{n-4}, \dots$  một lần mà là rất nhiều lần. Do đó, để khắc phục điều này thì giá trị nào tính rồi ta sẽ lưu vào một mảng mỗi lần sau chỉ cần tra mảng để lấy giá trị ra mà không phải tính lại, tiết kiệm được độ phức tạp về thời gian còn độ phức tạp về không gian thì tốn  $K(n) = \Theta(n)$ . Ban đầu mảng  $L$  sẽ khởi gán bởi những số 0 ý nói là chưa tính, giá trị nào tính rồi thì nó sẽ dương.

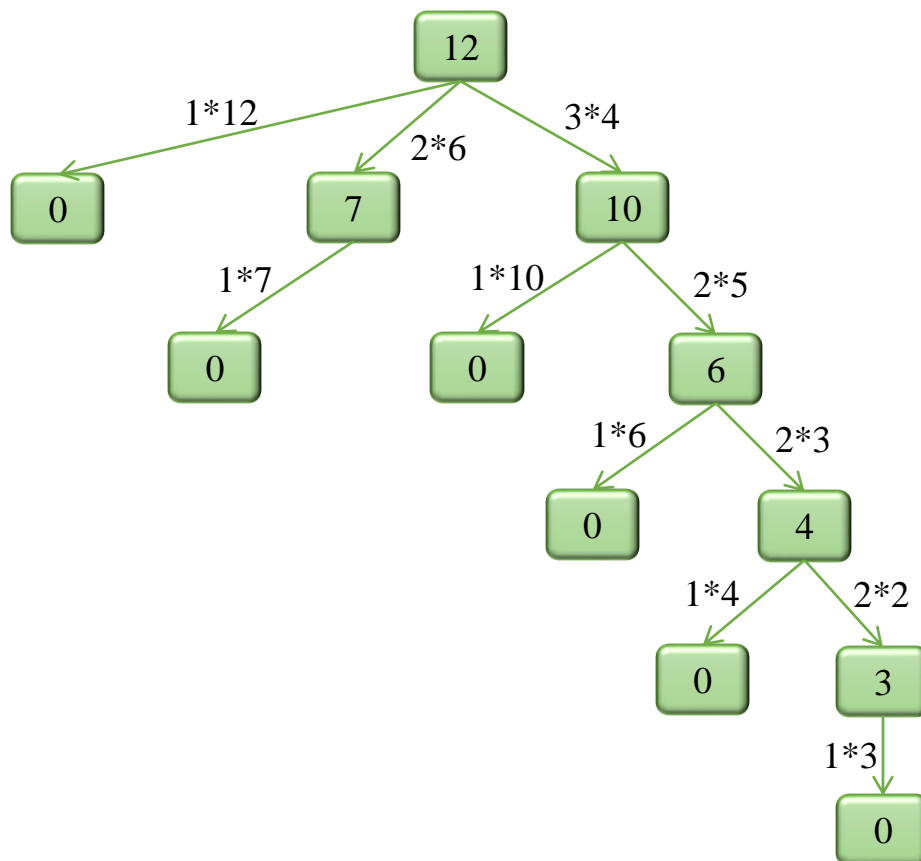
**Chương trình cải tiến:**

```
#include<stdio.h>
long *L;      //Bo nho luu cac gia tri tinh duoc
long LeoThang(int n)
{
    if(L[n]>0) return L[n]; //co roi khong tinh lai
    if(n==1) return L[n]=1;
    if(n==2) return L[n]=2;
    if(n==3) return L[n]=4;
    return L[n]=LeoThang(n-3)+LeoThang(n-2)+LeoThang(n-1);
}
int main()
{
    int n;
    printf("Nhap vao so bac n : "); scanf("%d",&n);
    L= new long [n+5];
    for(int i=1;i<=n;i++) L[i]=0;
    printf("So cach leo thang la %ld",LeoThang(n));
    delete []L;
}
```

**Đánh giá độ phức tạp của thuật toán cải tiến:** Mặc dù thuật toán chia để trị nhưng các nhánh sẽ được cắt tĩa hết cây đệ quy chỉ là một cây tuyến tính do đó độ phức tạp  $T(n) = \Theta(n)$ .

### 3.3.2. Bài toán mọi con đường về không

**Bài toán:** Một số nguyên dương  $n$  nếu phân tích được thành tích hai số nguyên dương  $n = a \times b$  trong đó  $a \leq b$  thì nó sinh ra số. Do  $a \leq b$  nên  $m = (a - 1) \times (b + 1) = ab - b + a - 1 = n - (b - a + 1) < n$ . Như vậy, số  $n$  có nhiều cách phân tích nên nó có thể sinh ra rất nhiều số  $m < n$  tiếp tục  $m$  lại sinh ra các số khác nhỏ hơn  $m$ , cứ như vậy tới cuối các bước sinh bao giờ cũng là số 0. Chẳng hạn số 12 có thể phân tích theo cây sau:



**Bài toán đặt ra:** Cho số nguyên dương  $n$  hỏi có bao nhiêu con đường đi đến số 0. Trong ví dụ trên thì số 12 có 6 cách đi đến 0.

**Phân tích thuật toán chia để trị:** Chúng ta thấy số 12 có 3 cách đi trực tiếp tới 0, 7 và 10 đây là những bài toán bé hơn, chúng ta sẽ dùng đệ quy để giải những bài toán này sau đó tính tổng lại để được bài toán ban đầu. Tổng quát hóa bài toán từ số  $n$  chúng ta duyệt tìm các ước của  $n$  để phân tích số và tìm ra những số mà từ  $n$  sinh ra trực tiếp rồi dùng đệ quy để tính từ những số này, sau đó tính tổng các con đường để được kết quả từ  $n$ .

**Thiết kế thuật toán chia để trị:**

**Trường hợp bài toán suy biến:** Khi  $n = 0$  thì kết quả trả về là 1, điều này rất quan trọng bởi nếu bạn trả về 0 thì mọi  $n$  đều chỉ ra 0 mà thôi.

**Trường hợp bài toán chưa suy biến:** Khi  $n \geq 1$

+ **Chia:** Chia bài toán ban đầu thành nhiều bài toán con tùy theo số ước của  $n$ . Nếu có phân tích  $n = a \times b$  với  $a \leq b$  chứng tỏ  $a \leq \sqrt{n}$  do

đó ta duyệt  $a$  chạy từ 1 đến  $\sqrt{n}$ , nếu  $n$  chia hết cho  $a$  thì sẽ sinh ra số  $m = (a - 1) \times (n / a + 1)$ .

+ **Trị:** Gọi đệ quy tính kết quả các bài toán con với những số  $m$

+ **Liên kết:** Dem tính tổng kết quả của các bài toán con được kết quả bài toán ban đầu.

### Chương trình

```
#include<stdio.h>
long Zero(int n)
{
    if(n==0) return 1;
    long d=0;
    for(int a=1;a*a<=n;a++)
        if(n%a==0) d+=Zero((a-1)*(n/a+1));
    return d;
}
int main()
{
    int n;
    printf("Nhap vao so bac n : "); scanf("%d",&n);
    printf("So con duong den 0 la %ld",Zero(n));
}
```

**Đánh giá độ phức tạp:** Đặt  $T(n)$  là độ phức tạp của thuật toán. Trong trường hợp tốt nhất  $n$  là số nguyên tố thì  $T(n) = \Omega(1)$ . Trong trường hợp xấu nhất thì  $n$  có  $\sqrt{n}$  ước số do đó  $T(n) = \sqrt{n}T(n-1) + c$  suy ra  $T(n) = O(n\sqrt{n}) = O(n^{3/2})$ . Thực ra số ước số tối đa không bao giờ lên tới  $\sqrt{n}$  và giá trị tối đa được sinh ra là  $n - 1$ , nhưng có rất nhiều số rất bé thậm chí sinh luôn ra số 0 và dừng, nên việc đánh giá cận trên này cũng có thể chưa thật sát.

**Cải tiến thuật toán thông qua sử dụng bộ nhớ trung gian:** Chúng ta thấy rằng có nhiều kết quả phải tính lặp đi lặp lại với chương trình như sau:

```
#include<stdio.h>
long *M;
long Zero(int n)
```

```
{
    if(M[n]>0) return M[n];
    if(n==0) return M[n]=1;
    for(int a=1;a*a<=n;a++)
        if(n%a==0) M[n]+=Zero((a-1)*(n/a+1));
    return M[n];
}
int main()
{
    int n;
    printf("Nhap vao so bac n : "); scanf("%d",&n);
    M=new long [n+5];
    for(int i=0;i<=n;i++) M[i]=0;
    printf("So con duong den 0 la %ld",Zero(n));
}
```

**Đánh giá độ phức tạp của thuật toán:** Chúng ta đặt  $T(n)$  là độ phức tạp của thuật toán. Trường hợp tốt nhất thì  $n$  nguyên tố và  $T(n) = \Omega(1)$ . Trong trường hợp xấu nhất ta phải đi tính tất cả các con đường về 0 của tất cả các số từ 0 đến  $n$ , nhưng vì có bộ nhớ quản lý nên việc tính toán chỉ thực hiện có 1 lần, còn những lần sau chỉ cần tra mảng  $M$  với độ phức tạp  $O(1)$  do đó tổng tất cả lại thì  $T(n) = O(n)$ .

### 3.3.3. Bài toán tìm dãy con liên tục có tổng lớn nhất

**Bài toán:** Cho dãy số nguyên  $a_1, a_2, \dots, a_n$  có  $n$  phần tử, một dãy con liên tục từ vị trí  $i$  đến vị trí  $j$  ( $1 \leq i \leq j \leq n$ ) là  $a_i, a_{i+1}, \dots, a_j$  có trọng số chính là tổng các giá trị của nó. Bài toán đặt ra là nhập vào dãy số nguyên và tìm trọng số của dãy con liên tục có trọng số lớn nhất.

**Chú ý:** Chúng ta thấy rằng nếu dãy toàn phần tử không âm thì dãy con liên tục có trọng số lớn nhất chính là cả dãy đó, còn nếu dãy toàn những phần tử âm thì dãy con liên tục có trọng số lớn nhất là giá trị lớn nhất của dãy.

**Ví dụ:** Với dãy -4, 7, -2, 3, -1, -2, 5, -8, 0, 1, -6, 3 thì trọng số lớn nhất cần tìm là 10 của dãy con 7, -2, 3, -1, -2, 5.

**Phân tích ý tưởng chia để trị:** Tổng quát hóa bài toán, chúng ta tìm dãy con liên tục có tổng lớn nhất của dãy  $a_L, a_{L+1}, \dots, a_R$  nếu ta đặt  $M = \lfloor (L + R) / 2 \rfloor$  khi đó có ba trường hợp:

- Thứ nhất là dãy con cần tìm nằm trọn nửa bên trái  $a_L, a_{L+1}, \dots, a_M$ , được bài toán nhỏ hơn, chúng ta dùng đệ quy để tìm;
- Thứ hai là dãy con cần tìm nằm trọn nửa bên phải  $a_{M+1}, a_{M+2}, \dots, a_R$  được bài toán nhỏ hơn, chúng ta dùng đệ quy để tìm;
- Thứ ba là dãy con cần tìm nằm trùm lên hai nửa việc này phải duyệt và tự giải

Chúng ta chỉ việc so sánh kết quả ba trường hợp để tìm ra đáp số.

**Thiết kế thuật toán chia để trị:**

**Trường hợp bài toán suy biến:** Khi  $L = R$  thì dãy chỉ có một phần tử nên trọng số lớn nhất cần tìm chính là  $a_L$ .

**Trường hợp bài toán chưa suy biến:** Khi  $L < R$

- + **Chia:** Đặt  $M = \lfloor (L + R) / 2 \rfloor$ , chúng ta chia bài toán ban đầu thành 2 bài toán con nhỏ hơn có cùng cách giải.



- + **Trị:** Gọi đệ quy tính kết quả nửa trái  $a_L, a_{L+1}, \dots, a_M$  và nửa phải  $a_{M+1}, a_{M+2}, \dots, a_R$ .

- + **Liên kết:** Xử lý trường hợp dãy con trùm lên hai nửa và so sánh kết quả với hai trường hợp đệ quy ở trên để tìm ra kết quả cần tìm.



Để được dãy con có trọng số lớn nhất trùm lên cả hai nửa, chúng ta thấy rằng cần phải tìm hai vị trí  $i$  và  $j$  sao cho  $L \leq i \leq M < M + 1 \leq j \leq R$  mà tổng

$a_i + \dots + a_M + a_{M+1} + \dots + a_j$  đạt giá trị lớn nhất. Chúng ta tách bài toán thành hai phần:

- Phần thứ nhất: tìm  $i$  sao cho  $L \leq i \leq M$  sao cho phần trái  $a_i + \dots + a_M$  lớn nhất bằng cách cộng cuốn chiếu từ  $a_M$  ngược về  $a_L$  và so sánh để tìm tổng lớn nhất.
- Phần thứ hai: tìm  $j$  sao cho  $M+1 \leq j \leq R$  sao cho phần phải  $a_{M+1} + \dots + a_j$  lớn nhất bằng cách cộng cuốn chiếu từ  $a_{M+1}$  xuôi đến  $a_R$  và so sánh để tìm tổng lớn nhất.

### **Chương trình**

```
#include<iostream>
using namespace std;
long Max(long *a,int L,int R){
    if(L==R) return a[L];
    int M=(L+R)/2;
    long ML=Max(a,L,M);
    long MR=Max(a,M+1,R);
    long TL = a[M], s =a[M];
    for(int i = M-1;i>=L;i--)
    {
        s+=a[i];
        if(TL <s) TL = s;
    }
    long TR = a[M+1], p=a[M+1];
    for(int i=M+2;i<=R;i++)
    {
        p+=a[i];
        if(TR<p) TR = p;
    }
    long MM = TL+TR;
    return MM>ML && MM>MR ? MM:(ML>MR ? ML:MR);
}
int main()
{
    int n;
```

```
long *A;
cout<<"\nNhap so phan tu:";
cin>>n;
A = new long[n+5];
for(int i=1;i<=n;i++)
{
    cout<<"\nA["<<i<<"]="";
    cin>>A[i];
}
cout<<"\nDay con lien tuc co tong MAX la:
"<<Max(A,1,n);
}
```

**Đánh giá độ phức tạp của thuật toán:** Đặt  $T(n)$  là độ phức tạp của thuật toán khi tìm dãy con liên tục có trọng số lớn nhất của dãy có  $n$  phần tử. Chúng ta thấy rằng mỗi bước có hai lần gọi đệ quy với kích cỡ là  $n/2$ , ngoài ra còn hai vòng lặp mỗi vòng  $O(n/2)$  để tính dãy con lớn nhất trùm cả hai nửa nên

$$T(n) = \begin{cases} c_0 & \text{khi } n = 1 \\ 2T(n/2) + cn & \text{khi } n > 1 \end{cases}$$

Theo dụng định lý Master thu gọn ta suy ra  $T(n) = \Theta(n \log n)$ .

### 3.3.4. Bài toán đếm số nghịch thế

**Bài toán:** Cho dãy số nguyên  $a_1, a_2, \dots, a_n$  có  $n$  phần tử, một cặp nghịch thế  $a_i, a_j$  ( $1 \leq i < j \leq n$ ) là hai phần tử của dãy sao cho số đứng trước lại lớn hơn số đứng sau ( $a_i > a_j$ ). Bài toán đặt ra là đếm xem trong dãy có tất cả bao nhiêu cặp nghịch thế.

**Ví dụ:** Với dãy 4 7 2 8 4 8 3 2 4 thì có 18 cặp nghịch thế sau:

Xét phần tử thứ nhất có 3 cặp: (4, 2), (4, 3), (4, 2);

Xét phần tử thứ hai có 5 cặp: (7, 2), (7, 4), (7, 3), (7, 2), (7, 4);

Xét phần tử thứ ba có 0 cặp;

Xét phần tử thứ tư có 4 cặp: (8, 4), (8, 3), (8, 2), (4, 4);

Xét phần tử thứ năm có 2 cặp: (4, 3), (4, 2);



Xét phần tử thứ sáu có 3 cặp : (8, 3), (8, 2), (8, 4);

Xét phần tử thứ bảy có 1 cặp : (3, 2);

Xét phần tử thứ tám có 0 cặp;

Xét phần tử thứ chín có 0 cặp.

**Phân tích ý tưởng chia để trị:** Chúng ta thực hiện ý tưởng chia để trị giống như phương pháp sắp xếp trộn thực hiện vừa sắp xếp vừa đếm số nghịch thế. Tổng quát hóa bài toán, chúng ta tìm số nghịch thế của dãy  $a_L, a_{L+1}, \dots, a_R$  nếu ta đặt  $M = \lfloor (L + R) / 2 \rfloor$  khi đó có ba trường hợp:

- Thứ nhất là số nghịch thế nằm bên trái  $a_L, a_{L+1}, \dots, a_M$ , được bài toán nhỏ hơn, chúng ta dùng đệ quy để tìm;
- Thứ hai là số nghịch thế nằm nửa bên phải  $a_{M+1}, a_{M+2}, \dots, a_R$  được bài toán nhỏ hơn, chúng ta dùng đệ quy để tìm;
- Thứ ba là số nghịch thế có phần tử nằm nửa bên trái lớn hơn phần tử nằm ở nửa bên phải,

Chúng ta tính tổng ba trường hợp để tìm ra tất cả số nghịch thế.

**Thiết kế thuật toán chia để trị:**

**Trường hợp bài toán suy biến:** Khi  $L = R$  thì dãy chỉ có một phần tử nên số nghịch thế là 0.

**Trường hợp bài toán chưa suy biến:** Khi  $L < R$

- + **Chia:** Đặt  $M = \lfloor (L + R) / 2 \rfloor$ , chúng ta chia bài toán ban đầu thành 2 bài toán con nhỏ hơn có cùng cách giải.



- + **Trị:** Gọi đệ quy tính kết quả nửa trái  $a_L, a_{L+1}, \dots, a_M$  và nửa phải  $a_{M+1}, a_{M+2}, \dots, a_R$ , đồng thời sắp xếp hai nửa đều không giảm.
- + **Liên kết:** Trộn hai dãy không giảm thành dãy không giảm đồng thời đếm số nghịch thế tạo bởi một phần tử bên trái ghép với một phần

từ bên phải dãy cộng với số nghịch thế hai nửa đếm được thành tất cả số nghịch thế của dãy.

**Minh họa:** Xét dãy 4 7 2 8 4 8 3 2 4 tách thành 2 nửa: nửa trái 4 7 2 8 4 và nửa phải 8 3 2 4.

- Khi gọi chương trình đệ quy: Chúng ta đếm số nghịch thế nửa trái là 4 cặp và sắp lại thành 2 4 4 7 8 và đếm số nghịch thế nửa phải là 4 cặp và sắp lại thành 3 2 4 8.
- Liên kết: Chúng ta trộn hai dãy tăng thành một dãy tăng bằng cách so sánh dần từ trái sang phải của hai nửa, mỗi khi phần tử nửa trái bé hơn hoặc bằng phần tử nửa phải được lấy thì không tạo ra nghịch thế, ngược lại nếu phần tử nửa phải nhỏ hơn được lấy thì nửa trái còn bao nhiêu phần tử thì sẽ có bấy nhiêu nghịch thế.

<i>Nửa trái</i>	<i>Nửa phải</i>	<i>Phần tử chọn</i>	<i>Dãy được trộn</i>	<i>Số nghịch thế</i>	<i>Tổng nghịch thế</i>
2 4 4 7 8	2 3 4 8	2	2	0	0
4 4 7 8	2 3 4 8	2	2 2	4	4
4 4 7 8	3 4 8	3	2 2 3	4	8
4 4 7 8	4 8	4	2 2 3 4	0	8
4 7 8	4 8	4	2 2 3 4 4	0	8
7 8	8	4	2 2 3 4 4 4	2	10
8	8	7	2 2 3 4 4 4 7	0	10
	8	8	2 2 3 4 4 4 7 8	0	10
		8	2 2 3 4 4 4 7 8 8	0	10

Như vậy số nghịch thế giữa phần tử nửa trái với nửa phải là 10, mà trong riêng nửa trái có 4, nửa phải có 4 vậy tổng số nghịch thế là 18.

### **Chương trình**

```
#include<iostream>
```

```
using namespace std;
int x[100005]; //mang trung gian de tron
long NghichThe(int *a, int L, int R)
{
    if(L==R) return 0;
    int M=(L+R)/2;
    long d = NghichThe(a,L,M) + NghichThe(a,M+1,R);
    int i=L,j=M+1;
    for(int p=L; p<=R; p++)
        if(i<=M && j<= R) //ca hai daycon
        {
            if(a[j]<a[i]) d+=M-i+1;
            x[p]=a[i]<=a[j]?a[i++]:a[j++];
        }
        else x[p]=i<=M?a[i++]:a[j++];
    for(int p=L;p<=R;p++) a[p]=x[p];
    return d;
}
int main()
{
    int n,*A;
    cout<<"\nNhap so phan tu:";    cin>>n;
    A=new int[n+5];
    for(int i=1; i<=n; i++)
    {
        cout<<"\nA["<<i<<"]=";
        cin>>A[i];
    }
    cout<<"\nSo nghich the la:"<<NghichThe(A,1,n);
}
```

**Đánh giá độ phức tạp của thuật toán:** Đặt  $T(n)$  là độ phức tạp của thuật toán khi đếm số nghịch thế trên dãy có  $n$  phần tử. Chúng ta thấy rằng mỗi bước có hai lần gọi đệ quy với kích cỡ là  $n/2$ , ngoài ra còn hai vòng lặp mỗi vòng  $O(n)$

để trộn và đếm số cặp nghịch thế mà phần tử lớn ở nửa trái, phần tử bé ở nửa phải.

$$T(n) = \begin{cases} c_0 & \text{khi } n = 1 \\ 2T(n/2) + cn & \text{khi } n > 1 \end{cases}$$

Theo dụng định lý Master thu gọn ta suy ra  $T(n) = \Theta(n \log n)$ .

### 3.3.5. Bài toán đổi tiền

**Bài toán:** Cho  $n$  mệnh giá tiền là các số nguyên dương  $a_1, a_2, \dots, a_n$  khác nhau từng đôi một, mỗi loại mệnh giá đều có vô hạn số tờ tiền. Muốn đổi số tiền  $M$  sang các mệnh giá đó sao cho số tờ tiền là ít nhất. Hãy lập trình nhập vào các số nguyên dương  $n, M$  và các số nguyên dương  $a_1, a_2, \dots, a_n$  khác nhau từng đôi một in ra số tờ tiền ít nhất đổi được

**Ví dụ:** Với ba mệnh giá tiền 3, 4, 1 muốn đổi số tiền  $M = 14$  thì số tờ tiền ít nhất là 4 gồm hai tờ tiền mệnh giá 3, hai tờ tiền mệnh giá 4 và không tờ tiền mệnh giá 1.

**Phân tích ý tưởng chia để trị:** Với số tiền  $M$  muốn đổi ra các mệnh giá  $a_1, a_2, \dots, a_n$  có số lượng tờ tiền mỗi loại là vô hạn. Chúng ta xét một số trường hợp:

- Nếu  $M < a_n$  thì mệnh giá  $a_n$  không có ý nghĩa chúng ta đệ quy giảm đề trị về bài toán nhỏ hơn sử dụng  $n - 1$  mệnh giá  $a_1, a_2, \dots, a_{n-1}$ .
- Nếu  $M \geq a_n$  thì có hai cách đổi tiền hoặc là không dùng mệnh giá  $a_n$  thì đệ quy về bài toán nhỏ hơn sử dụng  $n - 1$  mệnh giá  $a_1, a_2, \dots, a_{n-1}$  hoặc là có dùng mệnh giá  $a_n$  được một tờ cộng với đệ quy về bài toán nhỏ hơn là cách đổi số tiền  $M - a_n$  sử dụng các mệnh giá  $a_1, a_2, \dots, a_n$ . Chúng ta so sánh hai cách này cách nào ít hơn thì lấy.

**Thiết kế thuật toán chia để trị:** Không phải là bài toán luôn có lời giải, những lúc không tìm được cách đổi thì số tờ tiền chúng ta coi là dương vô cùng.

**Trường hợp bài toán suy biến:** Khi  $M = 0$  thì số tờ là 0.

**Trường hợp bài toán suy biến:** Khi  $M > 0$  và  $n = 0$  thì số từ  $\infty$ .

**Trường hợp bài toán chưa suy biến:** Khi  $n > 0$  và  $0 < M < a_n$  đưa về bài toán giảm đề trị gọi đệ quy đổi số tiền  $M$  với  $n - 1$  mệnh giá  $a_1, a_2, \dots, a_{n-1}$ .

**Trường hợp bài toán chưa suy biến:** Khi  $n > 0$  và  $M \geq a_n$  ta có.

+ **Chia:** Không làm gì cả

+ **Trị:** Gọi đệ quy giải bài toán con đổi số tiền  $M$  với  $n - 1$  mệnh giá  $a_1, a_2, \dots, a_{n-1}$  được số từ  $t_1$  và gọi đệ quy giải bài toán con đổi số tiền  $M - a_n$  với các mệnh giá  $a_1, a_2, \dots, a_n$  được số từ  $t_2$ .

+ **Liên kết:** So sánh  $t_1$  với  $1 + t_2$  để tìm giá trị nhỏ nhất ra kết quả bài toán.

**Chương trình:** Khi lập trình chúng ta sẽ coi  $M + 1$  là dương vô cùng, vì số tiền  $M$  mà đổi ra  $M + 1$  từ là không có cách đổi.

```
#include<iostream>
using namespace std;
int VC; //Gia tri duong vo cung
int DoiTien(int n,int *a,int M)
{
    if(M==0) return 0;
    if(n==0) return VC;
    if(M<a[n]) return DoiTien(n-1,a,M);
    int t1=DoiTien(n-1,a,M);
    int t2=1+DoiTien(n,a,M-a[n]);
    return t1<t2?t1:t2;
}
int main()
{
    int n,M,*a;
    cout<<"\nNhap so phan tu:";    cin>>n;
    a=new int[n+5];
    for(int i=1; i<=n; i++)
```

```
{
    cout<<"\na["<<i<<"]=";
    cin>>a[i];
}
cout<<"Nhập số tiền cần đổi M = "; cin>>M;
VC=M+1;
int t=DoiTien(n,a,M);
if(t>M) cout<<"\nKhông đổi được\n";
else cout<<"\nSố tờ tiền ít nhất là:"<<t;
}
```

**Đánh giá độ phức tạp của thuật toán:** Đặt  $T(n, M)$  là độ phức tạp của thuật toán khi đổi tiền sử dụng  $n$  mệnh giá. Khi đó gọi đệ quy hai lần  $n$  và  $n - 1$  mệnh giá nên  $T(n, M) = T(n - 1, M) + T(n, M - a[n]) + c$ , đây là cây nhị phân đầy đủ nên trong trường hợp xấu nhất  $T(n, M) = O(M \times 2^n)$ .

**Cải tiến thuật toán thông qua sử dụng bộ nhớ trung gian:** Độ phức tạp của thuật toán lớn như vậy bởi vì nhiều trường hợp ta phải tính đi tính lại rất nhiều lần. Để không phải tính lại chúng ta cải tiến thuật toán dùng thêm một mảng hai chiều lưu các giá trị tính được. Ban đầu các giá trị chưa tính lưu là  $-1$ , sau đó mỗi khi muốn tính ta tra bảng xem nếu có rồi giá trị sẽ không âm thì lấy ra dùng còn chưa có (giá trị là  $-1$ ) thì ta phải tính.

```
#include<iostream>
using namespace std;
int VC; //Giá trị dương vô cùng
int **C; //Bảng nhớ trung gian để lưu các giá trị tính được
int DoiTien(int n,int *a,int M)
{
    if(M==0) return C[n][M]=0;
    if(n==0) return C[n][M]=VC;
    if(C[n][M]!=-1) return C[n][M]; //tính rồi
    if(M<a[n]) return C[n][M]=DoiTien(n-1,a,M);
    int t1=DoiTien(n-1,a,M);
    int t2=1+DoiTien(n,a,M-a[n]);
```

```
        return C[n][M]=t1<t2?t1:t2;
    }
    int main()
    {
        int n,M,*a;
        cout<<"\nNhap so phan tu:";    cin>>n;
        a=new int[n+5];
        for(int i=1; i<=n; i++)
        {
            cout<<"\na["<<i<<"]=";
            cin>>a[i];
        }
        cout<<"Nhap so tien can doi M = "; cin>>M;
        VC=M+1;
        C=new int *[n+5];
        for(int i=0; i<=M; i++) C[i]=new int [M+5];
        for(int i=0; i<=n; i++)
        for(int j=0; j<=M; j++) C[i][j]=-1;
        int t=DoiTien(n,a,M);
        if(t>M) cout<<"\nKhong doi duoc\n";
        else cout<<"\nSo to tien it nhat la:"<<t;
    }
```

**Đánh giá độ phức tạp của thuật toán:** Chúng ta thấy rằng mỗi giá trị không phải tính đi tính lại mà chỉ là tra bảng nên độ phức tạp cả về thời gian và không gian là  $\Theta(M \times n)$ .

### 3.4. Thuật toán chia để trị trên dãy số

**Bài toán:** Nhập dãy số nguyên gồm  $n$  phần tử  $a_1, a_2, \dots, a_n$ , thực hiện một số nhiệm vụ sau:

- Xuất dãy vừa nhập ra màn hình
- Tìm giá trị nhỏ nhất trong dãy
- Tính tổng tất cả các giá trị của dãy
- Kiểm tra dãy có đơn điệu tăng không?

- Đếm số cặp 2 số liên tiếp mà số sau lớn hơn số trước
- Kiểm tra dãy có là dãy đối xứng không?

### 3.4.1. Thuật toán giảm để trị giải quyết bài toán

**Phân tích ý tưởng giảm để trị:** Để làm việc với dãy có  $n$  phần tử thì chúng ta đệ quy giảm để trị làm việc với dãy có  $n - 1$  phần tử, sau đó chúng ta chỉ cần giải quyết thêm phần tử thứ  $n$  là xong.

**Trường hợp bài toán suy biến:** Dãy có 0 phần tử hoặc 1 phần tử thì tự giải.

**Trường hợp bài toán chưa suy biến:** Ta gọi đệ quy giải bài toán với  $n - 1$  phần tử sau đó xử lý nốt phần tử thứ  $n$ .

### Chương trình

```
#include<stdio.h>
void Nhap(int n,int *a)
{
    if(n==0) return;
    Nhap(n-1,a);
    printf("a[%d] = ",n);
    scanf("%d",&a[n]);
}
void Xuat(int n,int *a)
{
    if(n==0) return;
    Xuat(n-1,a);
    printf("%8d",a[n]);
}
int Min(int n,int *a)
{
    if(n==1) return a[1];
    int m=Min(n-1,a);
    return m<a[n]?m:a[n];
}
int Tong(int n,int *a)
{
    return n==0?0:Tong(n-1,a)+a[n];
}
```



```
bool KiemTraTang(int n,int *a)
{
    if(n<2) return true;
    return a[n-1]<a[n] && KiemTraTang(n-1,a);
}
int DemCapSo(int n,int *a)
{
    return n<2?0:DemCapSo(n-1,a)+(a[n-1]<a[n]);
}
bool KiemTraDoiXung(int L,int R,int *a)
{
    if(L>=R) return true;
    return a[L]==a[R] && KiemTraDoiXung(L+1,R-1,a);
}
int main()
{
    int n,A[100005];
    printf("Nhap so phan tu n = ");
    scanf("%d",&n);
    Nhap(n,A);
    printf("\nDay vua nhap \n");
    Xuat(n,A);
    printf("\nGia tri nho nhat : %d",Min(n,A));
    printf("\nTong day so : %d",Tong(n,A));
    printf(KiemTraTang(n,A)?"\nDay don dieu tang":"\nDay
khong don dieu tang");
    printf("\nSo cap 2 so lien tiep so sau lon hon so
truoc %d",DemCapSo(n,A));
    printf(KiemTraDoiXung(1,n,A)?"\nDay doi xung":"\nDay
khong doi xung");
}
```

Độ phức tạp của mỗi thao tác đều là  $T(n) = T(n-1) + c = \Theta(n)$ , riêng kiểm tra đối xứng  $T(n) = T(n-2) + c = \Theta(n/2) = \Theta(n)$ .

**3.4.2. Thuật toán chia để trị giải quyết bài toán**

**Phân tích thuật toán chia để trị:** Để làm việc với dãy có  $n$  phần tử, chúng ta tổng quát hóa bài toán từ chỉ số  $L$  đến chỉ số  $R$  của dãy  $a_L, a_{L+1}, \dots, a_R$ . Chúng ta chia dãy thành hai nửa bởi điểm chia  $M = (L + R) / 2$ , sau đó gọi đệ quy giải quyết từng nửa và liên kết kết quả lại với nhau.

**Trường hợp bài toán suy biến:** Dãy có không phần tử ( $L > R$ ) hoặc dãy có 1 phần tử ( $L = R$ ) thì tự giải.

**Trường hợp bài toán chưa suy biến:** Ta gọi đệ quy giải 2 bài toán tương ứng nửa trái của dãy  $a_L, a_{L+1}, \dots, a_M$  và nửa phải của dãy  $a_{M+1}, a_{M+2}, \dots, a_R$  sau đó liên kết kết quả lại được kết quả bài toán lớn.

**Chương trình**

```
#include<stdio.h>
void Nhap(int L,int R,int *a)
{
    if(L==R)
    {
        printf("a[%d] = ",L);
        scanf("%d",a+L);
        return;
    }
    Nhap(L,(L+R)/2,a);
    Nhap((L+R)/2+1,R,a);
}
void Xuat(int L,int R,int *a)
{
    if(L==R) {printf("%8d",a[L]);return;}
    Xuat(L,(L+R)/2,a);
    Xuat((L+R)/2+1,R,a);
}
int Min(int L,int R,int *a)
{

```

```
    if(L==R) return a[L];
    int m1=Min(L,(L+R)/2,a);
    int m2=Min((L+R)/2+1,R,a);
    return m1<m2?m1:m2;
}
int Tong(int L,int R,int *a)
{
    return
L==R?a[L]:Tong(L,(L+R)/2,a)+Tong((L+R)/2+1,R,a);
}
bool KiemTraTang(int L,int R,int *a)
{
    if(L>=R) return true;
    return a[(L+R)/2]<a[(L+R)/2+1] &&
    KiemTraTang(L,(L+R)/2,a) &&
    KiemTraTang((L+R)/2+1,R,a);
}
int DemCapSo(int L,int R,int *a)
{
    if(L>=R) return 0;
    return DemCapSo(L,(L+R)/2,a)+
    DemCapSo((L+R)/2+1,R,a)+(a[(L+R)/2]<a[(L+R)/2+1]);
}

int main()
{
    int n,A[100005];
    printf("Nhap so phan tu n = ");
    scanf("%d",&n);
    Nhap(1,n,A);
    printf("\nDay vua nhap \n");
    Xuat(1,n,A);
    printf("\nGia tri nho nhat : %d",Min(1,n,A));
    printf("\nTong day so : %d",Tong(1,n,A));
    printf(KiemTraTang(1,n,A)?"\nDay don dieu tang":"\nDay
```

```
khong don dieu tang");  
    printf("\nSo cap 2 so lien tiep so sau lon hon so  
truong %d", DemCapSo(1, n, A));  
}
```

**Đánh giá độ phức tạp của thuật toán:** Đặt  $T(n)$  là độ phức tạp của thuật toán ta có  $T(n) = 2T(n/2) + c$ , theo định lý Master thu gọn  $T(n) = \Theta(n)$ .

### 3.5. Bài tập

**Bài 1.** Một nhóm gồm  $n$  người có thể tách ra thành hai nhóm nếu tồn tại cách tách hơn kém nhau đúng  $k$  người. Bài toán đặt ra là từ  $n$  người hỏi sau khi tách hết ra thì được bao nhiêu nhóm.

**Bài 2.** Cho một số có nguyên dương  $n$  chữ số, và một số nguyên dương  $k$  sao cho  $1 \leq k < n \leq 10^5$ . Bạn hãy lập trình xóa đi đúng  $k$  chữ số của số có  $n$  chữ số đó để được số còn lại giữ nguyên thứ tự sau khi xóa là bé nhất có thể.

**Bài 3.** Cho số 3 số nguyên  $a, b, c$  ( $a, c \neq 0$ ) hãy lập trình tính  $a^b \bmod c$ .

**Bài 4.** Cho số nguyên dương  $n$  ( $1 \leq n \leq 10^9$ ) tìm số Fibonacci thứ  $n$  nhưng số có thể quá lớn ta chỉ lấy phần dư khi chia  $F_n$  cho  $10^9 + 7$ .

**Bài 5.** Giả sử ta có một mảng gồm  $n$  phần tử và các phần tử có thể trùng nhau. Thiết kế giải thuật loại bỏ tất cả các phần tử trùng nhau chỉ giữ lại một phần tử.

**Bài 6.** Cho hình chữ nhật  $n \times m$ , một cánh tay Robot có thể cắt ngang hoặc dọc theo các cạnh của hình chữ nhật cách mép trên hoặc mép phải tương ứng, mỗi lần cắt hãy chỉ ra kích thước hình chữ nhật con lớn nhất trong các hình chữ nhật con thu được. Đầu vào là hai số nguyên dương  $n, m$  là kích thước hình chữ nhật, số nguyên dương  $k$  là số lần cắt và mỗi lần cắt là một cặp gồm kí tự 'N' nếu cắt ngang hoặc 'D' nếu cắt dọc và một số nguyên tương ứng với vị trí cắt, nếu cắt ngang thì cách mép trên còn cắt dọc thì cách mép trái một giá trị nguyên

duong. Đầu ra chỉ ra những diện tích hình chữ nhật lớn được lần lượt mỗi lần cắt trong tất cả k lần.

**Bài 7.** Nam có một dãy số nguyên dương n phần tử  $a_1, a_2, \dots, a_n$  sau đó ai đó đã bổ sung thêm một số nào đó vào vị trí bất kỳ trong dãy. Bạn hãy giúp Nam chỉ ra phần tử thêm vào đó với độ phức tạp thời gian là  $\log n$ .

**Bài 8.** Một con ếch muốn nhảy qua con suối có n hòn đá xếp thẳng hàng và con ếch đang đứng ở hòn đá thứ nhất nó phải nhảy đến hòn đá thứ n, mỗi lần nhảy ếch có thể nhảy sang hòn đá kế tiếp hoặc có thể bỏ qua đúng một hòn đá. Bạn hãy lập trình đếm xem ếch có bao nhiêu cách nhảy để đến được hòn đá thứ n.

**Bài 9.** Cho số nguyên dương n và số nguyên k ( $0 \leq k \leq n$ ) chúng ta biết rằng công thức tính tổ hợp chập k của n được tính như sau:

$$C_n^k = \begin{cases} 1 & \text{Khi } k = 0, n \\ C_{n-1}^{k-1} + C_{n-1}^k & \text{Khi } 1 \leq k < n \end{cases}$$

Hãy viết chương trình chia để trị có cải tiến thuật toán dùng bộ nhớ để tính tổ hợp chập k của n.

**Bài 10.** Bài toán mọi con đường về 0, cho hai số nguyên dương n, m ( $n > m$ ) xây dựng thuật toán chia để trị tìm số đường đi từ n đến m.

**Bài 11.** Bài toán mọi con đường về 0, cho hai số nguyên dương n, m ( $n > m$ ) xây dựng thuật toán giảm để trị xác định xem từ n có đi được tới m không?.

**Bài 12.** Bài toán dãy con liên tục có tổng lớn nhất, hãy chỉ ra vị trí đầu và cuối của dãy con tìm được, trong trường hợp có nhiều dãy có tổng lớn nhất bằng nhau thì chỉ ra những vị trí đầu cuối nhỏ nhất.

**Bài 13.** Cho dãy số nguyên  $a_1, a_2, \dots, a_n$  có n phần tử, một cặp thuận thể  $a_i, a_j$  ( $1 \leq i < j \leq n$ ) là hai phần tử của dãy sao cho số đứng trước lại nhỏ

hơn số đứng sau ( $a_i < a_j$ ). Bài toán đặt ra là đếm xem trong dãy có tất cả bao nhiêu cặp thuận thế.

**Bài 14.** Cho dãy số nguyên  $n$  phân tử  $a_1, a_2, \dots, a_n$  các phân tử đã được sắp xếp không giảm và một số nguyên  $x$ , hãy tìm số nguyên lớn nhất không vượt quá  $x$  ở dãy số đã sắp trên.

**Bài 15.** Cho  $n$  mệnh giá tiền là các số nguyên dương  $a_1, a_2, \dots, a_n$  khác nhau từng đôi một, mỗi loại mệnh giá đều có đúng một tờ tiền. Muốn đổi số tiền  $M$  sang các mệnh giá đó sao cho số tờ tiền là ít nhất. Hãy lập trình nhập vào các số nguyên dương  $n, M$  và các số nguyên dương  $a_1, a_2, \dots, a_n$  khác nhau từng đôi một in ra số tờ tiền ít nhất đổi được, trong trường hợp không đổi được in ra -1.

**Bài 16.** Cho  $n$  đồ vật có kích thước tương ứng là  $a_1, a_2, \dots, a_n$  và giá trị tương ứng là  $b_1, b_2, \dots, b_n$ . Một ba lô có kích thước  $M$ , bỏ qua hình dạng các đồ vật, bạn hãy xếp các đồ vật vào ba lô sao cho tổng kích thước không vượt quá kích thước ba lô nhưng tổng giá trị thu được là lớn nhất, in ra tổng giá trị lớn nhất thu được.

**Bài 17.** Cho đồ thị  $G = (V, E)$  có hướng, không có trọng số. Nhập vào hai đỉnh khác nhau  $s$  và  $f$  của đồ thị, hỏi có đường đi từ  $s$  đến  $f$  không?