Kiểm tra 1 số nguyên tố

+ **Định nghĩa**: Là số nguyên lớn hơn 1, chỉ có 2 ước là 1 và chính nó. Các số nguyên tố từ 1-100: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97

+ **Thuật toán**: Để kiểm tra 1 số nguyên n có phải số nguyên tố hay không, ta làm theo các bước.
- Nếu n<2 thì không phải số nguyên tố.
- Kiểm tra trong đoạn từ 2..sqrt(n) xem có ước của n không, nếu có tồn tại thì n không phải số nguyên tố
- Ngược lại, n là số nguyên tố.

+ **Code**:

Visual C# Code:

```
public bool isPrimeNumber(int n)
{
    if (n < 2) return false;
    int temp = (int)Math.Sqrt(n);
    for (int i = 2; i <= temp; i++)
    {
        if (n % i == 0) return false;
    }
    return true;
}
```

C++ Code:

```
bool isPrimeNumber(int n)
{
     if (n<2) return false;
     int temp=(int)sqrt(n);
     for (int i=2;i<=temp;i++)
       if (n%i==0) return false;
     return true;
}
```

Phương pháp sàng Eratosthene

+ **Mục đích**: Để lập bảng các số nguyên tố nhỏ hơn hoặc bằng 1 số n cho trước.

+ **Thuật toán**: Sử dụng 1 bảng bool isPrimeNumber[0..n+1] để lưu kết quả
- Khởi tạo: tất cả các số từ 1->n là nguyên tố.
- Xóa số 1 ra khỏi bảng.
- Lặp: Tìm 1 số nguyên tố đầu tiên trong bảng sau đó xóa tất cả các bội của nó trong bảng.
- Quá trình lặp kết thúc khi gặp số nguyên tố >= sqrt(n).
- Tất cả các số chưa bị xóa trong bảng là số nguyên tố.

+ **Code**:

Visual C# Code:

```
public bool[] SeiveOfEratosthene(int n)
{
    bool[] isPrime=new bool[n+1];
    //khởi tạo: các số từ 1->n được coi là nguyên tố.
    for (int i = 0; i <= n; i++)
    {
        isPrime[i] = true;
    }
    isPrime[1] = false;
```

```
    int k = 1;
    while (k <= (int)Math.Sqrt(n))
    {
        k++;
        //Tìm số nguyên tố đầu tiên
        while (!isPrime[k]) k++;
        //Xóa các bội của k khỏi danh sách các số nguyên tố
        int j = 2;
        while (k * j <= n)
        {
            isPrime[k * j] = false;
            j++;
        }
    }
    return isPrime;
}
```

Tìm ước số chung lớn nhất (GCD)

+ **Định nghĩa**: Ước số chung lớn nhất (Greatest Common Divisor) của 2 số a và b, được định nghĩa như sau: gcd(a,b)=d <=> d là số lớn nhất trong tất cả các ước chung của a và b.

+ **Thuật toán**: Giải thuật Euclid, định nghĩa đệ qui như sau:
- gcd(a,0)=a
- gcd(a,b)=gcd(b,a mod b)

+ **Code**:
-Đệ qui:
Visual C# Code:

```
public int GCD(int a, int b)
{
    if (b == 0) return a;
    else return GCD(b, a % b);
}
```

C++ Code:

```
int GCD(int a,int b)
{
    if (b==0) return a;
    else return GCD(b,a%b);
}
```

- Khử đệ qui:
Visual C# Code:

```
public int GCD(int a, int b)
{
    while (b != 0)
    {
        int temp = a % b;
        a = b; b = temp;
    }
    return a;
}
```

C++ Code:

```
int GCD(int a, int b)
{
    while (b != 0)
    {
        int temp = a % b;
        a = b; b = temp;
```

```
    }
    return a;
}
```

Tìm bội số chung nhỏ nhất (lcm)

+ **Định nghĩa**: Bội số chung nhỏ nhất (Least Common Multiple) của 2 số a và b được định nghĩa LCM(a,b)=m <=> m là số nhỏ nhất trong tất cả các bội chung của a và b.

+ **Thuật toán**: Ta có công thức:

+ **Code**:

Visual C# Code:

```
public int LCM(int a, int b)
{
    return Math.Abs(a * b) / GCD(a, b);
}
```

C++ Code:

```
int LCM(int a, int b)
{
    return abs(a * b) / GCD(a, b);
}
```

Tính n giai thừa (Factorial)

+ **Công thức**: . Phát biểu bằng lời: Giai thừa của 1 số nguyên dương n là tích tất cả các số nguyên dương nhỏ hơn hoặc bằng n.

+ **Thuật toán**: Ta có thuật toán đệ qui như sau:

+ **Code**:

- Đệ qui:

Visual C# Code:

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

- Khử đệ qui:

Visual C# Code:

```
public long Factorial(int n)
{
    long result = 1;
    for (int i = 1; i < n+1; i++)
    {
        result *= i;
    }
    return result;
}
```

C++ Code:

```
long Factorial(int n)
{
    long result = 1;
    for (int i = 1; i < n+1; i++)
    {
        result *= i;
    }
    return result;
}
```

Tính tổ hợp chập k của n

+ **Công thức**:

+ **Thuật toán**:
- Trâu bò (Brute Force): sử dụng hàm tính giai thừa.

- Sử dụng vòng lặp:

+ **Code**:

- Cơ bắp:
Visual C# Code:

```
public long binomialCoefficient(int n,int k)
{
    return Factorial(n) / (Factorial(k) * (Factorial(n - k)));
}
```

- Trí tuệ:
Visual C# Code:

```
public long binomialCoefficient(int n, int k)
{
    long res = 1;
    for (int i = 1; i <= k; i++,n--)
    {
        res = res * n / i;
    }
    return res;
}
```

Thuật toán sắp xếp quick sort

+ **Giới thiệu**: Quick sort là 1 trong các thuật toán sắp xếp nhanh và đơn giản nhất, được đề xuất bởi C.A.R. Hoare năm 1962. Nó hoạt động dựa trên chiến lược đệ qui bằng cách chia-để-trị.

+ **Ý tưởng**: Dãy cần sắp xếp a sẽ được phân ra làm 2 phần, sao cho tất cả các phần tử b trong phần thứ nhất sẽ luôn nhỏ hơn mọi phần tử c trong phần thứ 2 (quá trình chia-divide), sau đó 2 phần này sẽ được sắp xếp bằng cách gọi đệ qui tới chính thủ tục sắp xếp này (quá trình xử lý từng phần-conquer). Sau đó sẽ kết hợp 2 phần này lai để được 1 dãy đã sắp xếp (kết hợp-combine). Xem hình

Bước đầu tiên của quá trình phân đoạn chính là chọn ra 1 phần tử x dùng để so sánh, tất cả các phần tử nhỏ hơn x sẽ được đặt vào phần 1, tất cả các phần tử lớn hơn x sẽ được đặt vào phần 2. Các phần tử bằng x sẽ được đặt ở giữa (lưu ý là các phần tử này đã ở "đúng" vị trí của nó trong mảng)

+ **Quá trình phân tách**:
Input: Dãy a0, ..., an-1 có n phần tử

Output: Hoán vị các phần tử sao cho tất cả các phần tử của a0, ..., aj sẽ luôn nhỏ hơn các phần tử của ai, ..., an-1 (i>j)

Phương pháp:
Code:

```
        Chọn 1 phần tử x ở giữa làm phần tử so sánh.
        Gán i=0 và j=n-1
        while (i<=j)

            - Tìm phần tử ai đầu tiên lớn hơn hoặc bằng x

            - Tìm phần tử aj đầu tiên nhỏ hơn hoặc bằng x

            if (i<=j)
            đổi chỗ ai và aj
            gán i=i+1, j=j-1
```

Sau khi phân đoạn, giải thuật sẽ áp dụng đệ qui tương tự quá trình trên đối với 2 phần vừa tạo ra, quá trình đệ qui này sẽ kết thúc khi đoạn cần sắp xếp chỉ có duy nhất đúng 1 phần tử.

+ **Code**:

Visual C# Code:

```
void quicksort (int[] a, int lo, int hi)
{
    //  lo is the lower index, hi is the upper index
    //  of the region of array a that is to be sorted
    int i=lo, j=hi, h;

    // comparison element x
    int x=a[(lo+hi)/2];

    //  partition
    do
    {
        while (a[i]<x) i++;
        while (a[j]>x) j--;
        if (i<=j)
        {
            h=a[i]; a[i]=a[j]; a[j]=h;
            i++; j--;
        }
    } while (i<=j);

    //  recursion
    if (lo<j) quicksort(a, lo, j);
    if (i<hi) quicksort(a, i, hi);
}
```

+ **Đánh giá**: Trường hợp tốt nhất cho thuật toán quick sort, đó là mỗi lần gọi đệ qui sẽ chia ra được 2 phần có độ dài bằng nhau, để sắp xếp n phần tử, độ phức tạp thời gian cho nó sẽ là Θ(n log(n)). Do "độ sâu" của lời gọi đệ qui là log(n) và tại mỗi mức sẽ có n phần tử được tác động (Hình a).

Trường hợp xấu nhất khi mỗi lần gọi đệ qui sẽ chia ra 2 phần có độ dài không cân bằng, có nghĩa là 1 phần chỉ chứa có 1 phần tử, các phần tử còn lại sẽ nằm trong phần 2 (Hình c). Khi đó "độ sâu" của lời gọi đệ qui sẽ là O(n-1) và thời gian chạy sẽ là Θ(n2).

Trường hợp trung bình của quá trình phân đoạn được chỉ ra trong hình b.

Việc chọn phần tử x để so sánh sẽ quyết định kết quả của quá trình phân đoạn. Giả sử chọn x là phần tử nhỏ nhất trong đoạn, thì ta sẽ rơi vào trường hợp xấu nhất. Vì vậy, sẽ tốt hơn nếu chọn phần tử x mang giá trị trung bình trong dãy.
Khi x là phần tử lớn thứ n/2 trong dãy (trung vị), thì quá trình phân đoạn là tối ưu.
Trên thực tế, có thể tính giá trị trung vị trong thời gian tuyến tính. Các biến dạng khác nhau của thuật toán quick sort này sẽ chạy trong khoảng thời gian O(n log(n)) ngay cả trong trường hợp xấu nhất.
Tuy nhiên, vẻ đẹp của thuật toán quick sort lại nằm trong chính sự đơn giản của nó, dạng đơn giản của quick sort sẽ chạy trung bình trong khoảng thời gian O(n log(n)). Mặc dù, trong trường hợp xấu nhất độ phức tạp của nó vẫn là Θ(n2).

Kiểm tra số chính phương

+ **Định nghĩa**: Số chính phương (SquareNumber) là số có căn bậc 2 là 1 số nguyên. Ví dụ: 4,9,100...

+ **Thuật toán**: Để kiểm tra n có phải số chính phương hay không ta lấy phần nguyên của căn bậc 2 của n rồi bình phương, sau đó so sánh với n.
Ví dụ: sqrt(4)=2.0 Ta thấy 2^2=4 => 4 là số chính phương
sqrt(7)=2.4657 , phần nguyên là 2. Ta thấy 2^2 <> 7 => 7 không phải số chính phương.

+ **Code**:

Visual C# Code:

```
public bool isSquareNumber(int n)
{
    int temp = (int)Math.Sqrt(n);
    return (temp * temp == n);
}
```

Tìm kiếm nhị phân (Binary Search)

+ **Sơ lược**: Tìm kiếm nhị phân là thuật toán nhằm xác định vị trí của 1 phần tử trong 1 mảng đã được sắp xếp. Thuật toán hoạt động dựa trên việc so sánh giá trị phần tử đầu vào với phần tử ở giữa (middle) của dãy. Việc so sánh sẽ cho biết phần tử ở giữa này bằng, nhỏ hơn hay lớn hơn giá trị đầu vào. Khi phần tử được đem so sánh mà bằng input thì việc tìm kiếm sẽ kết thúc và trả về vị trí của phần tử đó. Nếu phần tử này nhỏ hơn input thì đồng nghĩa với việc ta cần tìm input trong đoạn từ middle+1..top, ngược lại nếu lớn hơn input thì ta cần tìm input trong đoạn bottom..middle-1. Khi dãy tại bước hiện tại không có phần tử nào (bottom>top) thì không cần tìm kiếm. Kết thúc quá trình tìm kiếm nếu không thấy input xuất hiện trong dãy thì kết quả trả về -1.

+ **Code**:

Visual C# Code:

```
public int BinarySearch(int[] a, int value, int bottom, int top)
{
    while (bottom <= top) // trong khi dãy đang xét còn phần tử
    {
        int middle = (bottom + top) / 2;
        if (a[middle] == value) return middle;  //  đã tìm thấy tại vị trí middle, quá trình tìm kiếm dừng.
        else
            if (a[middle] > value) top = middle - 1;    // cần tìm value trong đoạn bottom..middle-1
            else bottom = middle + 1;                   // cần tìm value trong đoạn middle+1..top
    }
```

```
    return -1;  //Không tìm thấy
}
```

Chú ý:
+ Độ phức tạp của thuật toán là O(log(n))
+ Thông thường giá trị bottom và top ban đầu tương ứng với 0 và N-1.

Số hoàn hảo (Perfect Number)

+ **Định nghĩa**: Số hoàn hảo là số có tổng các ước nhỏ hơn nó bằng chính nó. Ví dụ: 6 = 1+2+3. 28 = 1+2+4+7+14.

+ **Thuật toán**: Để kiểm tra n có phải Perfect Number hay không, đi tìm tổng tất cả các ước nhỏ hơn n rồi so sánh.

+ **Code**:

C++ Code:

```
bool isPerfectNumber(int n)
{
    int sum = 0;
    for (int i = 1; i <= n/2; i++)
    {
        if (n % i == 0) sum += i;
    }
    return sum == n;
}
```

Kiểm tra số nguyên tố cùng nhau

+ **Tác giả**: Peterdrew, link tham khảo http://forums.congdongcviet.com/showthread.php?t=37060 (postcount==8)

+ **Đặt vấn đề**: Trong khi thao tác với các con số nhiều lúc các bạn gặp phải cụm từ hai số nguyên tố cùng nhau!; lúc đó có bạn nghĩ rằng đó phải là 2 số nguyên tố gần nhau. Hì, sai mất rồi đó; vậy nên trong Vấn đề 5 Peter đã khái quát qua (dẫn các bạn trước khi vào vấn đề này) là: Hai số nguyên dương a và b được gọi là nguyên tố cùng nhau khi và chỉ khi Ước chung lớn nhất của chúng là 1; tức là (a,b)=1; chứ không hẳn a và b phải là 2 số nguyên tố đứng gần nhau (hoặc có một cái hiểu tương tự), dĩ nhiên nếu a và b là hai số nguyên tố khác nhau thì chúng cũng thoả mãn tính chất là hai số "nguyên tố cùng nhau" (Lẫn lộn quá, nhưng các bạn chú ý cho điều này).

+ **Phương pháp**: Vậy làm sao để kiểm tra chúng nguyên tố cùng nhau hay không? Rất đơn giản là (ký hiệu hàm là CoPrime()):
- Nếu ước chung lớn nhất của chúng khác 1 thì trị trả về cho hàm kiểm tra là 0, báo hiệu hai số không nguyên tố cùng nhau.
- Ngược lại, thì trị trả về cho hàm kiểm tra là 1, báo hiệu hai số nguyên tố cùng nhau.

+ **Code**:

C++ Code:

```
int CoPrime(int m,int n)
{
    return gcd(m,n)==1?1:0;
}
```

+ **Chú ý**: Bạn nào học lý thuyết mật mã rồi, chắc hẳn là hiểu cái kiểm tra này quan trọng thế nào, dù chỉ có 1 dòng code thế kia thôi.

Số các ước số nguyên dương của một số nguyên dương

+ **Tác giả**: Peterdrew, link tham khảo http://forums.congdongcviet.com/showthread.php?t=37060 (postcount==9)

+ **Đặt vấn đề**: Trong toán lý thuyết số thì vấn đề về liệt kê các ước nguyên dương của một số nguyên dương chiếm một vị trí cũng khá quan trọng, số các ước số nguyên dương của một số

nguyên dương n được cho bởi công thức sau:
Các bạn thường gặp một bài toán phổ biến là Phân tích một số ra tích các thừa số nguyên tố (mà ngày xưa học lớp 6 Peter mới được tiếp cận); nhìn thấy các bạn giải quyết chúng khá đơn giản bằng các vòng lặp for. Đó là:

+ **Code**:

C++ Code:

```
int NoOfDivisor(int n)
{
    int dem=0;
    for (int i=1;i<=(int)sqrt(n);i++)
        if (n%i==0)
            dem++;
    return dem;
}
```

Thuật toán Knuth-Morris-Pratt

+ **Tác giả**: Chuyên gia khoa học máy tính NQH

+ **Đặt vấn đề**: Pattern matching là một trong những bài toán cơ bản và quan trọng nhất của ngành máy tính.

Tìm patterns trong các chuỗi-DNA là bài toán cơ bản của sinh tin học. Các phần mềm quét virus hiện đại có mấy chục triệu "patterns" là các "chữ ký" (virus signature) của các con virus máy tính đã biết. Khi quét máy thì phần mềm phải tìm các patterns này trong các files hay bộ nhớ của máy. Mỗi pattern thường là một chuỗi bytes nhất định. Lệnh grep chúng ta thường dùng trong Unix cũng làm tác vụ tương tự, trong đó các patterns có thể được biểu diễn bằng các biểu thức chính quy (regular expressions).

Nếu chỉ tính các thuật toán tìm các xuất hiện của một chuỗi đơn (một chuỗi các ký tự cho sẵn) bên trong một chuỗi khác thôi thì ta đã có đến cả trăm thuật toán khác nhau. Knuth-Morris-Pratt (KMP) là một trong những thuật toán đó. KMP không phải là thuật toán nhanh nhất hay tốn ít bộ nhớ nhất trên thực tế. Trên thực tế các biến thể của thuật toán Boyer-Moore hay Rabin-Karp với hàm băm và bộ lọc Bloom thường được dùng. Ta sẽ nói về chúng sau. Nhưng KMP thật sự rất thanh lịch và nếu tác vụ tìm kiếm không ghê gớm quá thì KMP không kém Boyer-Moore là mấy.

Ý tưởng của KMP rất đơn giản. Nếu bạn đọc nó trong quyển CLRS (Introduction to Algorithms) thì bạn sẽ bị lạc trong sa mạc. Thật sự là CLRS làm cho mọi thứ phức tạp hơn cần thiết. Quá nhiều ký hiệu và quá ít trực quan. Ta sẽ thảo luận KMP dùng 2 bước. Bước 1 là thuật toán Morris-Pratt (1970, A linear pattern-matching algorithm, Technical Report 40, UC Berkeley). Bước 2 là một cải tiến của thuật toán MP, có thêm Knuth vào (1977, Fast pattern matching in strings, SIAM Journal on Computing 6(1):323-350).

Mặc dù hai bài báo cách nhau 7 năm, thật ra là thuật toán này đã được khám phá từ khoảng 1969 với một lịch sử thú vị. Phần cuối bài báo của KMP mô tả chi tiết lịch sử này. Cả phần kỹ thuật lẫn lịch sử trong bài báo đều rất chi tiết, kiểu Knuth, đọc rất thích.

1. Thuật toán Morris-Pratt
MP là thuật toán thời gian O(n) đầu tiên cho bài này. Ý tưởng của thuật toán MP là như sau. Giả sử ta muốn tìm pattern p[0..m-1] trong chuỗi s[0..n-1]. Đến một lúc nào đó thì ta có mis-match: s[j] != p[i] như trong hình sau:

Nếu dùng thuật toán cơ bắp (điều mà bạn nên làm khi đi phỏng vấn người ta hỏi viết strstr() lên bảng) thì ta dịch p một vị trí và làm lại từ đầu. Thời gian chạy sẽ là O(mn). Tồi! Nhưng mà làm lại từ đầu thì rất phí công chúng ta đã so sánh đến s[j]. Ta tìm cách dịch p đi xa hơn. Càng xa càng tốt miễn là không bị lỡ qua một xuất hiện của pattern trong chuỗi s. Dễ thấy rằng, để giữ vị trí của j không đổi thì ta phải dịch p đi một đoạn sao cho một tiền tố (prefix) của p[0..i-1] được xếp trùng bằng một hậu tố (suffix) của p[0..i-1], tại vì đoạn hậu tố này đã được so trùng với đoạn hậu tố cùng chiều dài của s[0..j-1]. Khi đó, ta chỉ cần tiếp tục so sánh s[j] với p[map[i]] mà không cần làm lại từ đầu. Trong đó, map[i] < i chỉ chỗ cho ta biết xê dịch p như thế nào.

Chiều dài của sự xê dịch (bằng với đại lượng i - map[i]) được gọi là một chu kỳ (period) của chuỗi p[0..i-1]. Phần tiền tố và hậu tố trùng khớp với nhau được gọi là biên (border) của chuỗi p[0..i-1]. Chiều dài của biên bằng i trừ đi chu kỳ. Để cho sự xê dịch có nghĩa, chu kỳ phải lớn hơn 0 và vì thế biên của p[0..i-1] luôn có chiều dài nhỏ hơn i. Để tránh trường hợp bị bỏ sót một xuất hiện của p trong s thì ta phải chọn biên dài nhất của p[0..i-1], ký hiệu là border(p[0..i-1]).

Trong thuật toán MP ta tính trước dãy map. Sau đó dùng dãy map này để so sánh hai chuỗi. Chặt chẽ hơn, với một chuỗi x bất kỳ, định nghĩa w = border(x) là chuỗi w dài nhất thỏa mãn $0 \le |w| < |x|$ sao cho w vừa là tiền tố vừa là hậu tố của x, nghĩa là x = wy = zw, trong đó y, z là các chuỗi nào đó. Lưu ý rằng $0 < |y| = |z| = \text{period}(x)$.

Bây giờ giả sử ta đã có bảng map[0...m], trong đó map[0] = -1 và $\text{map}[i] = |\text{border}(p[0..i-1])|$ với $1 \le i \le m$. Thuật toán Morris-Pratt có thể được viết như sau:

Python Code:

```
def MP(s, p): # print all occurrences of pattern p in string s
    map = compute_MP_map(p)
    i = 0
    n = len(s); m = len(p)
    for j in range(n):
        while ( (i >= 0) and (s[j] != p[i]) ):
            i = map[i]
        i = i+1
        if (i == m):
            print "Match at position ", j-m+1
            i = map[i]
```

Ta phân tích thời gian chạy của MP dùng phương pháp phân tích khấu hao (Amortized Analysis). Ta cho mỗi chú s[j] 2 đồng xu để dùng. Và tưởng tượng bọn p[i] là m cái rọ. Lúc đầu bỏ vào rọ p[0] một đồng xu làm vốn. Mỗi lần phép so sánh ký tự ở dòng 6 được chạy thì ta dùng đồng xu có sẵn trong rọ p[i] để trả nợ cho phép so sánh này. (Lưu ý rằng ta giả sử nếu i=-1 thì không có phép so sánh. Nếu bạn cẩn thận bạn có thể bẻ đôi điều kiện của while ra để tránh truy cập p[-1]. Đoạn Python trên tôi đã chạy, không có vấn đề gì.) Xong rồi đến cuối, ngay trước khi thực hiện phép gán i = i+1 ở dòng 8 ta lấy 2 đồng xu của s[j] bỏ vào p[i] và p[i+1]. Như vậy thời gian chạy của MP là O(n), và nó chỉ dùng nhiều nhất 2n phép so sánh.

Vấn đề tiếp theo là làm sao tính map. Đây là một bài toán quy hoạch động hay. Lưu ý rằng biên của biên của một chuỗi x cũng là biên của x. Giả sử ta muốn tính map[i+1] = |border(p[0..i])|. Dễ thấy rằng, nếu p[i] = p[map[i]] thì map[i+1] = map[i] + 1. Nếu p[i] != p[map[i]] thì ta so p[i] với p[map[map[i]]], vân vân. Từ đó ta có đoạn mã sau:

Python Code:

```
def compute_MP_map(p):
    m = len(p)          # p is the input pattern
    map = [-1]*(m+1)    # map[0..m], the Morris-Pratt border map
    i = 0; j = map[i]
    while (i < m):
        while ( (j >= 0) and (p[i] != p[j]) ):
            j = map[j]
```

```
        j = j+1; i = i+1
        map[i] = j
    return map
```

Bài tập: dùng phương pháp phân tích khấu hao tương tự như trên, chứng minh rằng thời gian chạy của compute_MP_map() là O(m).

Chạy thử:

C++ Code:

```
>>> MP("abcabcabcabc", "cabc")
Match at position  2
Match at position  5
Match at position  8
```

2. Thuật toán Knuth-Morris-Pratt

KMP cải tiến map một chút. Trong hình ở trên, nếu p[map[i]] = b thì ta lại có mis-match. Do đó, ta áp đặt một cú "dòm trước" (look ahead) khi tính map. Gọi MPmap là dãy map của thuật toán MP. Cái map cải tiến của thuật toán KMP được định nghĩa như sau. KMPmap[0] = -1. Với 1 ≤ i ≤ m-1 thì gọi j = MPmap[i]. Khi đó, nếu p[i] != p[j] thì KMPmap[i] = j. Nếu p[i] == p[j] thì KMPmap[i] = KMPmap[j]. Và cuối cùng, KMPmap[m] = MPmap[m]. Bạn nên nghĩ cẩn thận xem tại sao định nghĩa KMPmap như vậy là hữu lý.

Dĩ nhiên, nếu đã tính MPmap rồi thì tính KMPmap theo công thức trên dễ dàng thôi. Nhưng ta có thể viết hàm tính KMPmap trực tiếp. Đây là một bài tập lập trình rất hay! Trong Python có thể viết như sau:

Python Code:

```
def compute_KMP_map(p):
    m = len(p)          # p is the input pattern
    map = [-1]*(m+1)    # map[0..m], the Knuth-Morris-Pratt border map
    i = 1; map[i] = 0; j = map[i]
    while (i < m):
        # at this point, j is MP map[i], which is not necessarily KMP map[i]
        if (p[i] == p[j]):
            map[i] = map[j]
        else:
            map[i] = j
            while ( (j >= 0) and (p[i] != p[j]) ):
                j = map[j]
        j = j+1; i = i+1
    map[m] = j
    return map

def KMP(s, p): # print all occurrences of pattern p in string s
    map = compute_KMP_map(p)
    i = 0; j = 0
    n = len(s); m = len(p)
    for j in range(n):
        while ( (i >= 0) and (s[j] != p[i]) ):
            i = map[i]
        i = i+1
        if (i == m):
            print "Match at position ", j-m+1
            i = map[i]

>>> KMP("abcabcabcabc", "cabc")
Match at position  2
Match at position  5
Match at position  8
```

Điểm thú vị cuối cùng là, mỗi ký tự của chuỗi s được so sánh với nhiều nhất là (đại lượng này gọi là delay của thuật toán), trong đó là tỉ lệ vàng!

+ **Code hoàn chỉnh**:

Visual C# Code:

```
using System;

namespace Knuth_Morris_Pratt
{
    class Test
    {
        private char[] s;
        private char[] p;

        public void Nhap()
        {
            Console.Write("Nhap xau S:");
            string stdin1 = Console.ReadLine();
            s = new char[stdin1.Length];
            s = stdin1.ToCharArray();
            Console.Write("Nhap xau P:");
            string stdin2 = Console.ReadLine();
            p = new char[stdin2.Length];
            p = stdin2.ToCharArray();
        }

        private int[] compute_MP_map()    // map[i]=| border(p[0..i-1]) |
        {
            int m = p.Length;    // p is the input pattern
            int[] MP_map = new int[m + 1];     // lưu ý rằng ta phải tính
map[0..m]
            // init
            MP_map[0] = -1;     // cái này chắc ai cũng hiểu
            // bay gio di tinh map[1..m]
            int i = 0; int j = MP_map[i];
            while (i < m)
            {
                while (j >= 0 && (p[i] != p[j])) j = MP_map[j];
                j++; i++;
                MP_map[i] = j;
            }
            return MP_map;
        }

        private int[] compute_KMP_map() //thuật toán KMP cải tiến
        {
            int m = p.Length;
            int[] KMP_map = new int[m + 1];
            int[] MP_map = compute_MP_map();
            KMP_map[0] = -1; KMP_map[m] = MP_map[m];
            for (int i = 1; i < m; i++)
            {
                int j = MP_map[i];
                if (p[i] != p[j]) KMP_map[i] = j;
                else KMP_map[i] = MP_map[j];
            }
            return KMP_map;
        }

        public void Morris_Pratt()
        {
            int[] map = compute_KMP_map();
```

```
            int n = s.Length;
            int m = p.Length;
            int i = 0;
            string res = "";
            for (int j = 0; j < n; j++)
            {
                while ((i >= 0) && (p[i] != s[j])) i = map[i];
                i++;            // Có 2 khả năng xảy ra: hoặc là đã đi hết chuỗi p
(i=m-1) hoặc là i=-1 , lợi dụng cả 2 điều này
                if (i == m)
                {
                    res += (j - m + 1).ToString() + " ";
                    i = map[i];
                }
            }
            Console.Write(res);
        }


        static void Main(string[] args)
        {
            Test object1 = new Test();
            object1.Nhap();
            object1.Morris_Pratt();

            //Console.WriteLine("done");
            Console.ReadLine();
        }
    }
}
```

Liệt kê dãy nhị phân có độ dài n

+ **Ví dụ**: n=3 ta có các dãy nhị phân như sau

Code:

```
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
```

+ **Tác giả**: CayDan

+ **Code**:

- **Phương pháp sinh**:

C Code:

```
void LKNhiPhan(int n)
{
  int x[30];
  for (int i=0; i<n; i++)
        x[i] = 0;
  int i=n;
  while (i>=0)
  {
      for (i=0; i<n; i++)
         printf("%d  ",x[i]);
      printf("\n");
      i=n-1;
```

```
      while ((i>=0) && (x[i] == 1)) i--;
      if(i>=0)
      {
         x[i] = 1;
         for (int j= i+1; j<n; j++)
            x[j] = 0;
      }
   }
}
```

- **Phương pháp đệ qui**:

C Code:

```
int x[30];
void Try(int i, int n)
{
    for (int k=0; k<=1; k++)
    {
        x[i] = k;
        if (i==n-1)
        {
            for (int j=0;j<n;j++)
                printf("%d ",x[j]);
            printf("\n");
        }
        else   Try(i+1, n);
    }
}

void LKNhiPhan(int n)
{
   Try(0, n);
}
```

Đếm số phần tử khác nhau của mảng

+ **Tác giả**: XuyenIT55

+ **Ví dụ minh họa**:
{-3,0,2,-3,-3,2,1} => có -3,0,2,1 => kết quả = 4.

+ **Giải thuật**:
-Sắp xếp mảng.
-Kiểm tra nếu a[i]!=a[i-1] thì tăng biến đếm lên 1

+ **Code**:

C++ Code:

```
int Count(int a[], int n)
{
     if (n<0) return 0;
     sort(a,0,n-1);
     int dem=1;
     for ( int i = 1 ; i < n ; i++)
     {
         if ( a[i] != a[i-1])
              dem++;
     }
     return dem;
}
```

Kiểm tra 1 số nguyên tố lớn (Thuật toán Miller-Rabin)

+ **Tác giả**: hunterphu

+ **Code**:

C++ Code:

```
bool isprime(unsigned long long n)
{
	if(n==1) return false;
	for (unsigned long long i=2;i*i<=n;i++)
	    if(n%i==0) return false;
	return true;
}
unsigned long long power(unsigned long long base,unsigned long long exp,unsigned l
ong long mod)
{
	unsigned long long res=1;
	while (exp>0)
	{
		if (exp%2==1)
		   res=(res*base)%mod;
		exp/=2;
		base=(base*base)%mod;
	}
	return res;
}
bool miller_rabin(unsigned long long n,int t)
{
	if (n<=50) return isprime(n);
	unsigned long long m=n-1;
	int r,k=0;
	if (n%2==0)return false;
	while (m%2==0)
	{
		m=m/2;
		k++;
	}
	for (int i=0;i<t;i++)
	{
	    r=k;
	    unsigned long long a=rand()%(n-2)+2;
	    unsigned long long b=power(a,m,n);
	    if (b!=1&&b!=n-1)
	    {
	       while (r>=0)
	       {
	            b=power(b,2,n);
	            if(b==n-1)break;
	            r--;
	       }
	       if (r<0)return false;
	    }
	}
	return true;
}
```

Đếm số chữ số của số nguyên

+ **Ví dụ**: x=3022 => kết quả 4
x=140017 =>Kết quả 6

+ **Code**:

- Tác giả: Peterdrew

Cách 1:

C++ Code:

```
int dem=0;
while(n)
{
    dem++;
    n/=10;
}
```

Cách 2:

C++ Code:

```
floor(log10(abs(n!= 0?n:1)))+1;
```

Lưu ý: Phải include thư viện math.h

- Tác giả: Aydada

C++ Code:

```
int Count(int x)
{
    char a[33];//(sizeof(int)*8+1)

    itoa(x, a, 10)
    return strlen(a);
}
```

- Tác giả: Vietduc

C++ Code:

```
int so_chu_so(int n){
    return n? so_chu_so(n/10)+1 : 0;
}
```

Thuật toán kiểm tra số nguyên tố (Full + Advanced)

+ **Tác giả**: mp121209 && Rox_Rook

+ **Nội dung**: Kinh nghiệm cho thấy rằng ngôn ngữ lập trình, trong một giới hạn nào đó, là công cụ để chúng ta mô hình
và giải quyết các bài toán thực tiễn từ đơn giản nhất cho đến phức tạp nhất. "Bài toán" mà tôi muốn nhắc tới ở đây
bao gồm các bài toán trong toán học nói riêng, hoặc một "bài toán" nào đó, một công việc cụ thể nào đó trong thực tế.
Bài viết này sẽ giải quyết vấn đề "Kiểm tra tính nguyên tố của một số nguyên dương" và một vài vấn đề khác
liên quan đến nó.
Khái niệm toán học: (Định nghĩa số nguyên tố) Số nguyên tố là một số nguyên dương, chỉ có hai ước số phân biệt là
1 và chính nó. Từ định nghĩa chúng ta hãy phân tích đôi chút về tính chất của chúng. Số 1 có ước là 1 và chính nó,
nhưng trùng nhau, suy ra, 1 không phải là số nguyên tố. Số 2 là số nguyên tố chẵn duy nhất, và cũng là số nguyên
tố nhỏ nhất. Các số nguyên tố còn lại là số lẻ. Giả sử số nguyên tố là số chẵn, suy ra nó có dạng 2 * n, n là một số
tự nhiên lớn hơn 1. Nhưng 2 * n chia hết cho 2, suy ra, không phải nguyên tố.
Từ toán học đến tin học: Ta đã rõ tính chất của các số nguyên tố. Nhưng số nguyên tố dùng để làm gì? Vì sao

chúng ta lại cần tìm chúng? Chúng ta không nghiên cứu sâu về vấn đề này, mà giả sử rằng, ở một nơi nào đó số
nguyên tố đóng một vai trò hết sức quan trọng. Chẳng hạn, trong ngân hàng , chúng được sử dụng để mã hóa tài khoản,
... Và nhiệm vụ của bạn là phải tìm chúng, hoặc quy luật phát triển của chúng.
Đặt bài toán:
Đầu vào: Số nguyên dương N (Để đơn giản, giả sử N không quá lớn, kiểu int trong C/C++ đủ để hiện thực hóa).
Đầu ra: True – nếu N là số nguyên tố, False – nếu N không phải là số nguyên tố. (Để tiết kiệm không gian bộ nhớ
và giảm kích thước chương trình tôi dùng kiểu bool với hai giá trị là true (đúng), và false (sai) để lưu giá trị trả về của
việc check tính nguyên tố của số N).
Giải quyết bài toán:
Như vậy, để kiểm tra một số có là nguyên tố hay không, theo suy nghĩ trực quan của tất cả các lập trình viên,
thậm chí những người khác không hiểu gì về toán học và thuật toán, mà chỉ biết những thông tin trong phạm vi bài
viết này, thì chúng ta cần kiểm tra xem số đó có ước số nào khác ngoài 1 và chính nó hay không, nếu có thì đó là
hợp số (combine number) còn nếu không thì số đó là số nguyên tố.
Thuật toán đầu tiên và đơn giản nhất hiện ra trong đầu chúng ta lúc này có lẽ là: Kiểm tra các số có khả năng
là ước sốcủa N (số cần kiểm tra tính nguyên tố), các số này nằm trong đoạn [2,N-1]. Thuật toán được cài đặt đơn
giản như sau:

C++ Code:

```
// the first algorithm check a number to see
// if it is a prime number. we have to check
// all values in range [2,n-1] and assert if
// any of them is a divisor of n.
// input: a positive integer n
// output: true if n is prime, otherwise -false
bool check_prime (const int &n) {
    // 1 is not a prime
    if (n == 1)
        return false;
    // if we'll find any divisor of n, then n is a combine
    for (int i = 2; i < n; i ++)
        if (n % i == 0)
            return false;
    // in the end, n is a prime
    return true;
}
```

Còn có nhiều cách cài đặt tương đương, nhưng tôi chỉ đưa ra một cách điển hình. Ở cách cài đặt này, vòng
lặp for sẽ chạy từ 2 đến N – 1 để có thể đưa ra kết luận cuối cùng. Tuy nhiên, nếu suy nghĩ thêm một chút chúng
ta sẽ thấy rằng không cần thiết phải kiểm tra đến giá trị N – 1 mà thực chất chỉ cần tới [N/2] ([] là kí hiệu lấy
phần nguyên) vì không ước nào của n có thể lớn hơn [N/2] trừ chính nó! Thuật toán trên được cài đặt lại như sau:
C++ Code:

```
// the second algorithm
bool check_prime (const int &n) {
    // 1 is not a prime
    if (n == 1)
        return false;
    // if we'll find any divisor of n, then n is a combine
```

```
    // note, you should save n/2 in a temporary variable to
    // step up the computational process.
    // int m = n/2;
    // for (int i = 2; i <= m; i ++) {...}
    for (int i = 2; i < n / 2; i ++)
        if (n % i == 0)
            return false;
    // in the end, n is a prime
    return true;
}
```

Tuy nhiên, nếu suy nghĩ thêm một chút, chúng ta thấy rằng không cần phải kiểm tra đến N/2, mà chỉ cần đến
căn bậc hai của N, [sqrt(N)] là đủ. Tại sao lại như vậy? Rất đơn giản nếu các bạn chịu khó suy nghĩ một chút! Vì thế,
thuật toán 2 lại được cài đặt lại như sau:
C++ Code:

```
// the third algorithm
bool check_prime (const int &n) {
    // 1 is not a prime
    if (n == 1)
        return false;
    // if we'll find any divisor of n, then n is a combine
    for (int i = 2; i <= sqrtl(n); i ++)
        if (n % i == 0)
            return false;
    // in the end, n is a prime
    return true;
}
```

Xem ra có vẻ thuật toán của chúng ta đã tạm ổn, và chương trình đã chạy tương đối nhanh! Tuy nhiên, như
đã phân tích từ đầu, số nguyên tố là các số lẻ, trừ số 2. Vì thế chúng ta chỉ cần kiểm tra các ước số của N là các
số lẻ, nếu chúng không phải là ước số của N thì N chính là nguyên tố, ngược lại, là hợp số. Thuật toán 3 được cải
tiến thành thuật toán 4, như sau:
C++ Code:

```
// the fourth algorithm
bool check_prime (const int &n) {
    // 2 is prime
    if (n == 2)
        return true;
    // 1 is not a prime
    // each even number is not a prime
    if (n == 1 || n % 2 == 0)
        return false;
    // if we'll find any divisor of n, then n is a combine
    // you should save the value of sqrtl(n) in a temporary
    // variable, otherwise program will calculate sqrt and
    // call function sqrtl, it will take a long time ...
    long j = sqrtl(n);
    for (int i = 3; i <= j; i +=  2)
        if (n % i == 0)
            return false;
    // in the end, n is a prime
    return true;
}
```

Rõ ràng là so với thuật toán trước đó, số giá trị của i cần kiểm tra đã giảm đi một nửa. Như vậy, mấu chốt trong
việc tăng tốc và cải tiến thuật toán nằm ở câu lệnh thay đổi giá trị của biểu thức điều khiển biến i, ta cần giảm số giá
trị của i cần kiểm tra đến minimum thì sẽ đạt được hiệu quả maximum của thuật toán. Vừa rồi ta đã loại bỏ được một
nửa số giá trị của i bằng cách xét ước số 2 có thể của N. Tiếp theo ta sẽ cải tiến thuật toán bằng

cách xét ước số
nguyên tố tiếp theo của N là 3. Nếu N chia hết cho 2, hoặc 3 thì dễ dàng kết luận nó là hợp số (combine). Ngược lại,
N sẽ không chia hết cho 2 * 3 = 6. Vậy ước số của N sẽ có dạng 6 * i + 1, 6 * i + 2, 6 * i + 3, 6 * i + 4, 6 * i + 5. Để ý
rằng 6 * i + 2 và 6 * i + 4 là bội số của 2, còn 6 * i + 3 là bội số của 3. Vì thế, ước số có thể có của N chỉ còn lại 6 * i + 1,
hoặc 6 * i + 5, cách nhau 2 đơn vị! Ta sẽ bắt đầu với giá trị i = 5 và sẽ chọn công thức 6 * i + 5. Tuy nhiên nếu chỉ kiểm
tra i thì sẽ thiếu mất ứng cử viên ở dạng 6 * i + 1. Vì thế ta chỉ cần thêm i + 2 vào cho đủ. (Lưu ý, i = 6 * j + 5; vì thế ở
iteration tiếp theo ta có j = j + 1; i + 2 = 6 * (j + 1) + 2 = 6 * j + 7. Điều này cho thấy 6 * i + 1 và 6 * i + 4 cách nhau
2 đơn vị! Có người bảo rằng chúng cách nhau 4 đơn vị, nhưng dễ thấy 6 – 4 = 2. Để hiểu thêm về sự giải thích này, các
bạn có thể tìm đọc bài viết này của tác giả Rox_Rook.
Vì thế 4 đơn vị ở iteration trước sẽ còn lại 2 đơn vị ở iteration sau mà thôi!). Thuật toán mới sẽ như sau:
C++ Code:

```
// the fifth algorithm
bool check_prime (const int &n) {
    // 2 is prime, 3 is prime
    if (n == 2 || n == 3)
        return true;
    // 1 is not a prime
    // each even number is not a prime
    if (n == 1 || n % 2 == 0 || n % 3 == 0)
        return false;
    long j = sqrtl(n);
    for (int i = 5; i <= j; i +=  6)
        if (n % i == 0 || n % (i + 2) == 0)
            return false;
    // in the end, n is a prime
    return true;
}
```

Một số phân tích toán học đơn giản sẽ giúp chúng ta nhận thấy rằng nếu ban đầu chúng ta đặt i = 5; thì ở
iteration tiếp theo giá trị của i sẽ cách 5 đúng 2 đơn vị, tức là i += 2; ở iteration tiếp thì i lại tăng lên 4 đơn vị, như
đã phân tích ở bước trên. Ta có thể cài đặt lại thuật toán thứ 5 thành thuật toán 5', gọi là thuật toán nhảy cóc 2, 4
để kiểm tra số nguyên tố như sau. Lưu ý: Thuật toán 5' và thuật toán 5 trên thực tế là tương đương, không thuật
toán nào phát huy tính hiệu quả hơn!
C++ Code:

```
// the fifth' algorithm
bool check_prime (const int &n) {
    // 2 is prime, 3 is prime
    if (n == 2 || n == 3)
        return true;
    // 1 is not a prime
    // each even number is not a prime
    if (n == 1 || n % 2 == 0 || n % 3 == 0)
        return false;
    long j = sqrtl(n);
    int k = 2;
    for (int i = 5; i <= j; i +=  k, k = 6 - k)
        if (n % i == 0)
            return false;
    // in the end, n is a prime
    return true;
}
```

Trên thực tế, khi chúng ta phải làm việc với dữ liệu lớn về số lượng lẫn kích thước thì việc xử lý trên một đối tượng
thường lặp lại nhiều lần. Thuật toán tìm số nguyên tố cũng được cải thiện. Khi kiểm tra tính chất nguyên tố của một số,
người ta chỉ kiểm tra các ước số có thể, nhưng là nguyên tố (vì một ước số nếu là hợp số, thì nó lại chia hết cho một số
nguyên tố nhỏ hơn. Định lý toán học: Mọi hợp số đều có thể phân tích thành tích các thừa số nguyên tố). Vì vậy ta sẽ
sinh trước một mảng các số nguyên tố (về bản chất là đánh dấu các số đó là nguyên tố) không vượt quá giới hạn ước
số của N. Sau đó kiểm tra cá ước của n chỉ cần kiểm tra xem n có chia hết cho các số nguyên tố nhỏ hơn hoặc bằng
sqrt(N) có nằm trong mảng đã tạo không. Thuật toán này là thuật giải sàng Erastosthene.
Nếu các bạn quan tâm có thể tìm đọc, tôi không đề cập đến ở đây!
Từ việc phân tích một bài toán nhỏ ở trên, tôi hi vọng mang lại cho các bạn một sự lưu tâm, một chút suy nghĩ nào
đó. Bài viết trình bày theo cách xây dựng bài toán mang tính chất kế thừa và phát triển. Từ việc phân tích bài toán,
vẽ giản đồ sơ lược, đến cài đặt thuật giải và tối ưu nó. Đó cũng chính là tư tưởng cho việc tư duy để giải quyết mọi vấn
đề, cũng là đặc trưng cơ bản về tính kế thừa mà các bạn sẽ được làm quen, và phải hiểu rõ khi học tập và nghiên cứu C++.

Phân tích một số thành thừa số nguyên tố

+ **Problem**: Phân tích 1 số thành thừa số nguyên tố và đưa ra kết quả dưới dạng (m1, n1)(m2,n2)....
Trong đó m(i) là thừa số nguyên tố, còn n(i) là số mũ.
ví dụ:
18= (2,1)(3,2)
50= (2,1)(5,2)

+ **Written by**: mp121209

+ **Code**:
C++ Code:

```cpp
#include <iostream>
int main () {
    int n;
    std::cin >> n;
    int count = 0;
    for (int i = 2; i <= n; i ++) {
        while (n % i == 0) {
            count ++;
            n /= i;
        }
        if (count ) {
            std::cout << "(" << i << "," << count << ")" << std::endl;
            count = 0;
        }
    }
}
```

+ **Test Result**:
Code:

```
105
(3,1)
(5,1)
(7,1)
Press any key to continue . . .
```

+ **Proof**: Một số không phải là số nguyên tố luôn luôn biểu diễn được dưới dạng tích các thừa số là nguyên tố. Ta sẽ lần lượt chia số cần biểu diễn cho số nguyên tố nhỏ nhất là 2, sau đó tăng dần giá trị lên 3, 4, 5, 6. Tất nhiên bạn sẽ thấy 4, 6 ... không phải là số nguyên tố, nhưng thuật toán sẽ không sai, vì một số chia hết cho 4 khi và chỉ khi nó chia hết cho 2*2, tức là ta thực hiện vòng lặp chừng nào lần, đến khi giá trị của số bị chia n không còn chia hết cho số chia nữa thì mới tăng số chia i lên. Mỗi lần chia cho i, số bị chia sẽ giảm đi i lần, tức là giá trị của nó còn lại n/i. Lặp cho đến khi số chia to bằng số bị chia thì dừng Trường hợp này chỉ xảy ra nếu số bị chia là nguyên tố

_____

Code C

```
bool isPrime(int n)
{
if(n==2 || n==3 ) return true;

if(n%2 ==0 ) return false;

if(n%3 == 0 ) return false;

int i=5,j=2;

while(i*i <= n)
{
if(n%i == 0 ) return false;
i+= j;
j = 6-j;
}

}
```

Với thuât toán trên thì chương trình sẽ chạy nhanh hơn rất nhiều nhất là khi N là số nguyên tương đối lớn.