Chuyên đề:

SÀNG SỐ NGUYÊN TỐ CẢI TIẾN & ỨNG DỤNG

MỤC LỤC

Thầy/Cô có thể tải Test, Code mẫu theo link sau: http://bit.ly/2KcuxG4

1. Tóm tắt:

Số học hay còn gọi là lí thuyết số là một trong những ngành toán học cổ nhất của nhân loại. Theo thời gian đã có nhiều thuật toán về số được đề xuất giúp giải quyết các vấn đề về số học như kiểm tra số nguyên tố, tìm ước chung lớn nhất, mã hóa…Đây được xem là những thành tựu to lớn của nhân loại với sự góp mặt của các nhà toán học vĩ đại như: Euclid, Euler, Fermat…

Trong bồi dưỡng học sinh giỏi Tin học, số học giữ vai trò rất quan trọng, là kiến thức nền tảng không thể thiếu cho các em. Đặc biệt trong số học, sự xuất hiện của nhiều loại số khác nhau có những tính chất đặc biệt như Fibonacci, Catalan, Số hoàn hảo, Số nguyên tố… luôn chứa đựng các bí ẩn bên trong qui luật của nó. Ta thử tìm hiểu một vài điều thú vị về số nguyên tố.

Như ta đã biết, mọi số tự nhiên lớn hơn 1 đều có thể phân tích thành tích các số nguyên tố. Điều này cho thấy từ các số nguyên tố, ta có thể xây dựng nên toàn bộ các số tự nhiên. Bên cạnh đó, số nguyên tố chính là yếu tố quyết định trong hệ mã hóa công khai RSA được sử dụng rộng rải ngày nay. Số 113 của lực lượng cảnh sát cơ động cũng là số nguyên tố…

Trong một số bài toán, ta rất hay gặp các yêu cầu cần phải xác định được các số nguyên tố trong một giới hạn nào đó như: Liệt kê các số nguyên tố, tính tổng các số nguyên tố …. Với các thuật toán kiểm tra số nguyên tố theo định nghĩa ta không đủ thời gian để xử lý khi khoảng dữ liệu quá lớn. Vì thế, một nhóm thuật toán đã ra đời, giúp ta liệt kê danh sách các số nguyên tố trong đoạn [1, N] bằng cách kiểm tra khả năng nguyên tố của các số nguyên trong đoạn. Nhóm thuật toán này là các Sàng số nguyên tố.

Trong khuôn khổ chuyên đề, tôi xin trình bày các thuật toán về sàng số nguyên tố như: Eratosthenes, Atkin và Sundaram. Tôi cũng tiến hành so sánh hiệu năng của các thuật toán với nhau. Tiếp đến tôi sẽ thực hiện cải tiến thuật toán sàng Atkin, sàng Eratosthenes để mang lại hiệu suất cao hơn nhưng vẫn dễ cài đặt. Cuối cùng sẽ là một số các bài toán minh họa theo các mức độ khác nhau.

Chuyên đề hướng đến đối tượng là học sinh lớp 10. Do đó hướng đến các cách cải tiến có cài đặt không quá phức tạp để các em tiếp thu tốt. Giới hạn chuyên đề đạt được là $N = 10^8$. Các cách cài đặt tối ưu hơn nhưng phức tạp hơn để đạt được $N = 10^9$, 10^{10} sẽ được giới thiệu đến trong phần kết luận. Thầy cô đồng nghiệp và các bạn quan tâm có thể tìm hiểu thêm.

Cách thức triển khai giảng dạy:

1. Ta nhắc lại định nghĩa số nguyên tố và bài toán cần xét. Giới thiệu thuật toán vét cạn và chỉ ra nhược điểm của nó.
2. Giảng dạy cho học sinh kiến thức về từng loại sàng số nguyên tố. So sánh hiệu năng của chúng. Cho học sinh cài đặt nhuần nhuyễn các thuật toán sàng số nguyên tố cần thiết.
3. Cải tiến thuật toán Sàng Eratosthenes theo một số cách đơn giản, dễ cài đặt.
4. Cho bài tập áp dụng theo từng mức độ chủ yếu dùng sàng Eratosthenes để minh họa:
- Mức Cơ bản (Bài 1, 2, 3): chỉ áp dụng sàng số nguyên tố thông thường có biến đổi để giải quyết các bài toán thường gặp.
- Mức khá (Bài 4, 5): các bài toán bắt buộc phải áp dụng thuật toán cải tiến để xử lý, có kết hợp các yếu tố khác: tính tổng, xử lý xâu…
- Mức Khó (Bài 6): Bắt buộc phải áp dụng thuật toán cải tiến để hổ trợ. Nhưng phải có thuật toán thông minh để giải quyết vấn đề.
5. Tổng kết và nêu hướng phát triển cho học sinh. Các em sẽ được học ở giai đoạn sau.

2. Nội dung

2.1 Định nghĩa số nguyên tố:

Số tự nhiên N > 1, được gọi là số nguyên tố nếu N chỉ có đúng hai ước là 1 và chính nó.
Ví dụ: Số 11 là số nguyên tố do chỉ có 2 ước là 1 và 11. Số 9 không phải là số nguyên tố do có 3 ước là 1, 3, 9.

2.2 Bài toán

Nhận thấy Tom là một học sinh xuất sắc và bị hấp dẫn rất nhiều về số nguyên tố, Thầy giáo lại quyết định cho Tom một thử thách tiếp theo là tìm tổng của N số nguyên tố đầu tiên. Do giới hạn khá lớn nên Tom hơi bị lúng túng. Em hãy giúp anh ấy tìm cách giải bài toán này thật nhanh.

Input: file SUMNT.INP

- Dòng đầu tiên chứa số lượng các test **T**
- **T** dòng tiếp theo, mỗi dòng chưa số nguyên dương M

Output: file SUMNT.OUT

- Xuất ra **T** số nằm trên **T** dòng trả lời cho **T** test ở trên.

Ràng buộc:

$1 <= T <= 80$

$1 <= N <= 10^6$

Ví dụ:

SUMNT.INP	SUMNT.OUT
2 6 16	41 160

Chú ý: Số nguyên tố có thể lên đến 10^8

Giải thích

Ta có T = 2

Khi N = 6, tổng các số nguyên tố là = 2+3+5+7+11+13= 28

Khi N = 11, tổng các số nguyên tố là = 2+3+5+7+11+13+17+19+23+29+31= 160

- Để giải quyết bài toán trên ta cần phải liệt kê được danh sách các số nguyên tố từ đó tính tổng N số nguyên tố đầu tiên.
- Sau đây là 1 số thuật toán đánh dấu và đếm các số nguyên tố trong đoạn [1..N]

2.2. Thuật toán Vét cạn (Brute Forces)

Đây là thuật toán sử dụng kỹ thuật vét hết tất cả các số lẻ và kiểm tra tính nguyên tố của nó theo định nghĩa.

Code C++

```
#include <bits/stdc++.h>
using namespace std;
void Bruce_forces(long limit)
{
     long count = 1;
     bool isPrime = true;
     for (long i = 3; i <= limit; i += 2)
     {
          for (long j = 3; j * j <= i; j += 2)
          {
               if ((i % j) == 0) { isPrime = false; break; }
          }
          if (isPrime) { count++; }
          else isPrime = true;
     }
     cout<<count<<endl;
}
int main()
{
     long limit = 10000000;
     Bruce_forces(limit);
     return 0;
}
```

Độ phức tạp: O((n$\sqrt{n}$)/4)

2.2. Sàng Eratosthenes

Eratosthenes Cyrene (cổ Hy Lạp: 276 – 195 TCN) là Nhà toán học, địa lý, nhà thơ, vận động viên, nhà thiên văn học, sáng tác nhạc, nhà triết học. Eratosthenes là người đầu tiên tính ra chu vi trái đất và khoảng cách từ Trái đất tới Mặt trời với độ chính xác ngạc nhiên bằng phương pháp đơn giản nhất là đo bóng Mặt Trời tại hai địa điểm khác nhau trên Trái Đất.

Sàng Eratosthenes hoạt động theo tư tưởng loại bỏ dần bội số của các số nguyên tố thỏa một giới hạn nhất định. Khi kết thúc thuật toán, các số chưa bị loại bỏ chính là các số nguyên tố cần tìm.

Ta có thể liệt kê mọi số nguyên tố không quá 10^9 bằng sàng Eratosthenes, tuy nhiên chỉ hiệu quả khi $N<=10^7$

Chi tiết thuật toán [4]:

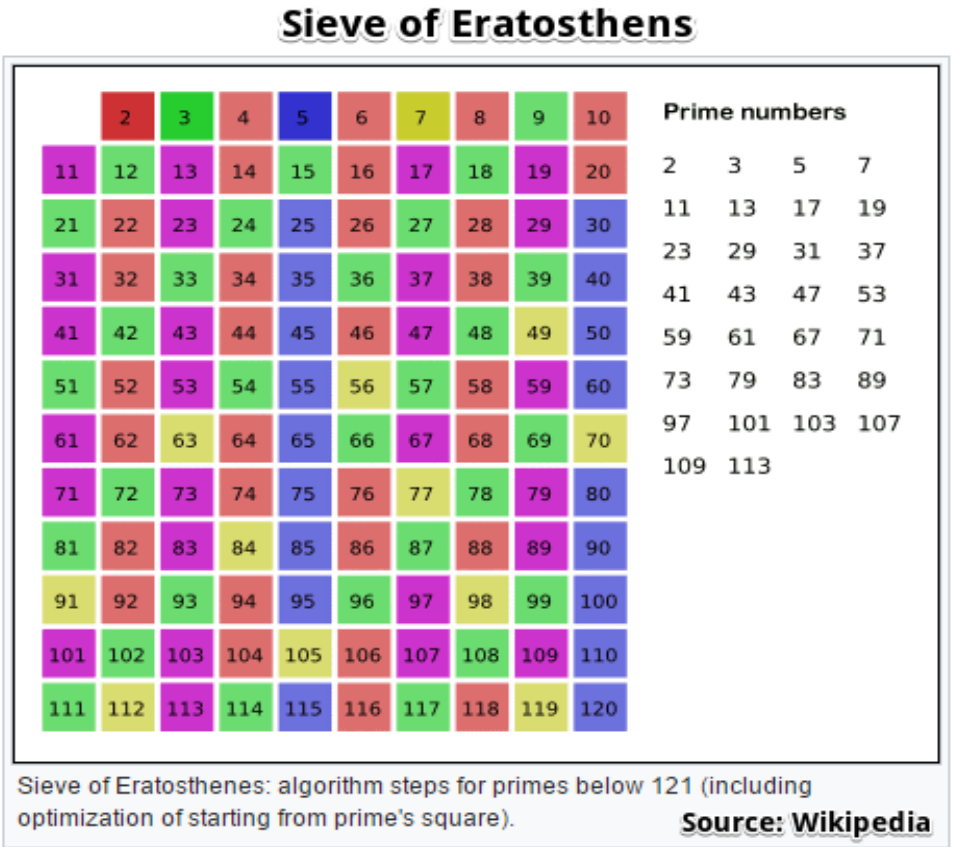
Bước 1: Tạo 1 danh sách các số tự nhiên liên tiếp từ 2 đến n: (2, 3, 4,..., n).

Bước 2: Giả sử tất cả các số trong danh sách đều là số nguyên tố. Trong đó, p = 2 là số nguyên tố đầu tiên.

Bước 3: Tất cả các bội số của p: 2p, 3p, 4p,... sẽ bị đánh dấu vì không phải là số nguyên tố.

Bước 4: Tìm các số còn lại trong danh sách mà chưa bị đánh dấu và phải lớn hơn p. Nếu không còn số nào, dừng tìm kiếm. Ngược lại, gán cho p giá trị bằng số nguyên tố tiếp theo và quay lại bước 3.

Khi giải thuật kết thúc, tất cả các số chưa bị đánh dấu trong danh sách là các số nguyên tố cần tìm.



Nhìn vào hình ta thấy: Các màu đậm là các số nguyên tố, cụ thể:

- Số 2 là số nguyên tố được tô màu đỏ, các bội của nó lần lượt là 4, 6, 8… bị loại được tô màu đỏ nhạt.
- Số 3 là số nguyên tố được tô màu xanh lá cây đạm, các bội của nó 9, 12, 15… bị loại được tô màu xanh lá cây nhạt.

- Tương tự cho số 5 và 7
- Các số màu hồng đậm còn lại chính là các số nguyên tố.

Code C++:

```
#include <bits/stdc++.h>
using namespace std;
bool prime[10000001];
void SieveOfEratosthenes(long n)
{
    memset(prime, true, sizeof(prime));
    for (long p=2; p*p<=n; p++)
    {
        // Nếu prime[p] không đổi, p là số nguyên tố
        if (prime[p] == true)
        {
            // Loại bỏ tất cả các bội của p
            for (long i=p*p; i<=n; i += p)
                prime[i] = false;
        }
    }
    long count=0;
    for (long p=2; p<=n; p++)
       if (prime[p])
          //cout << p << " ";
            count++;
    cout<<count<<endl;
}
int main()
{
    long n = 10000000;
    SieveOfEratosthenes(n);
    return 0;
}
```

Độ phức tạp:

Ta nhận xét vòng lặp for bên trong:

- ✓ Với i = 2, vòng lặp sẽ chạy N/2 lần
- ✓ Với i = 3, vòng lặp sẽ chạy N/3 lần
- ✓ Với i = 5, vòng lặp sẽ chạy N/5 lần

➔ Ta có tổng số lần thực hiện: $N * (1/2 + 1/3 + 1/5 + \ldots)$

➔ Độ phức tạp thuật toán: $O(n \log\log n)$[1]

[1] https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

2.3. Sàng Atkin[2]

Đây là một thuật toán nhanh và hiện đại để tìm tất cả các số nguyên tố thỏa một giới hạn nào đó. Thuật toán được tối ưu từ sàng Eratosthenes bằng cách đánh dấu các bội số của bình phương của các số nguyên tố chứ không phải là bội của các số nguyên tố. Thuật toán được xây dựng bởi A. O. L. Atkin và Daniel J. Bernstein.

Trong thuật toán có áp dụng phương pháp "Wheel factorization" (Bánh xe phân tích) giúp giảm đáng kể số lượng số cần xét, từ đó làm giảm lượng bộ nhớ lưu trữ.

- **Phương pháp Wheel factorization:**

Đây là một phương pháp giúp tạo ra một danh sách nhỏ các số "cận" nguyên tố từ các công thức toán học đơn giản. Danh sách này được sử dụng để làm tham số đầu vào cho các thuật toán khác nhau trong đó có sàng Atkin.

Các bước thực hiện[3]:

1) Tìm một vài số nguyên tố đầu tiên để làm cơ sở cho phương pháp.
2) Nhân các số nguyên tố cơ sở ở 1 lại với nhau được giá trị là n. n chính là chu vi của bánh xe phân tích.
3) Viết các số nguyên từ 1 đến n theo một vòng tròn. Đây là vòng tròn bên trong nhất thể hiện một vòng quay của bánh xe.
4) Với các số từ 1 đến n, ta đánh dấu bỏ các bội của các số nguyên tố cơ sở vì các số này không phải là số nguyên tố.
5) Gọi x là số lượng vòng quay hiện tại, ta tiếp tục viết các số từ x*n + 1 đến x*n + n theo các vòng tròn đồng tâm với vòng tròn ở 3. Sao cho số thứ x*n + 1 kề với số thứ (x + 1)*n
6) Lặp lại bước 5 cho đến khi đạt giới hạn cần kiểm tra.
7) Đánh dấu bỏ số 1
8) Đánh dấu bỏ các số là bội của các số nguyên tố cơ sở nằm trên cùng một phần rẽ quạt của bánh xe
9) Đánh dấu bỏ các số là bội của các số nguyên tố cơ sở nằm khác phần rẽ quạt với các số cơ sở, và trên các vòng tròn còn lại tương tự như bước 4.
10) Các số còn lại trên bánh xe số là các số "cận" nguyên tố. Nghĩa là đa số là các số nguyên tố, còn lại xen lẫn một vài số là hợp số. Ta có thể sử dụng các thuật toán để lọc lại như sàng Eratosthenes hay Atkin…

[2] https://en.wikipedia.org/wiki/Sieve_of_Atkin

[3] https://en.wikipedia.org/wiki/Wheel_factorization

Ví dụ minh họa:

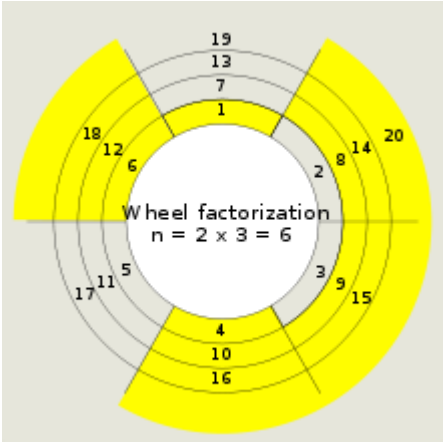
Áp dụng phương pháp Wheel factorization với n = 2x3 = 6

1) Hai số nguyên tố cơ sở: 2 and 3.

2) n = 2 × 3 = 6

3) Tạo vòng tròn các số: 1 2 3 4 5 6

4) Đánh dấy loại bỏ 4 và 6 vì là bội của 2; 6 là bội của 3:



1 2 3 4 5 6

5)

- x = 1.
- x*n + 1 = 1*6 + 1 = 7.
- (x + 1)*n = (1 + 1) * 6 = 12.

Viết các số từ 7 đến 12 vào vòng tròn thứ 2, với 7 thẳng hàng với 1.

1 2 3 4 5 6
7 8 9 10 11 12

6)

- x = 2.
- xn + 1 = 2 · 6 + 1 = 13.
- (x + 1)n = (2 + 1) · 6 = 18.

Viết các số từ 13 đến 18 vào vòng tròn thứ 3, với 13 thẳng hàng với 7

Lặp lại cho các vòng tròn tiếp theo, ta được

1 2 3 4 5 6
7 8 9 10 11 12
13 14 15 16 17 18
19 20 21 22 23 24
25 26 27 28 29 30

7 và 8) Loại bỏ các số là bội của 2, 3 trên cùng phần rẽ quạt

1 2 3 4 5 6
7 8 9 10 11 12
13 14 15 16 17 18
19 20 21 22 23 24
25 26 27 28 29 30

9) Loại bỏ các số là bội của 2, 3 trên tất cả các vòng tròn của bánh xe.

1 2 3 4 5 6
7 8 9 10 11 12
13 14 15 16 17 18
19 20 21 22 23 24
25 26 27 28 29 30

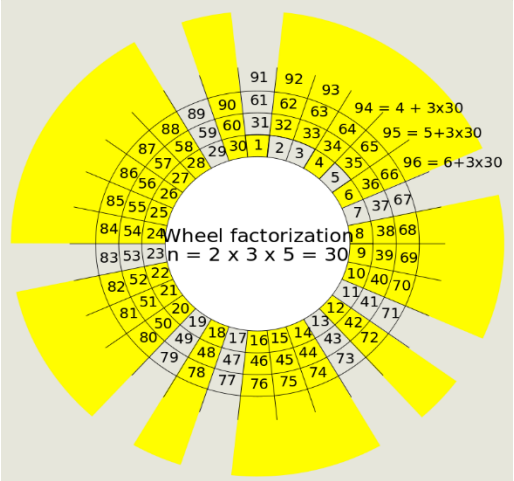
10) Danh sách kết quả chứa một số không phải là số nguyên tố là 25. Sử dụng các thuật toán khác để loại bỏ các phần tử như các sàng số nguyên tố

2 3 5 7 11 13 17 19 23 29

Trở lại thuật toán Sàng Atkin

Ta cần lưu ý một số điều trong thuật toán:

- Tất cả các số dư đều là số dư trong phép chia cho 60, nghĩa là chia cho 60 lấy dư
- Tất cả các số nguyên, kể cả x và y đều là các số nguyên dương
- Phép đảo số là chuyển trạng thái của một số từ không là số nguyên tố thành số nguyên tố và ngược lại.
- Bánh xe phân tích được sử dụng trong sàng Atkin là 2x3x5 = 30. Xét 2 vòng quay là 60.



Thuật toán:

1) Tạo bảng kết quả, điền vào 2, 3, và 5.
2) Tạo bảng sàng nguyên tố với các số nguyên dương; tất cả các số đánh dấu là không nguyên tố.
3) Với tất cả các số trong sàng:
    - Nếu số đó chia 60 dư 1, 13, 17, 29, 37, 41, 49, hoặc 53, đảo đánh dấu cho các số có dạng $4*x^2 + y^2$. **Hay số đó chia cho 12 dư 1 hoặc 5.**
    - Nếu số đó chia 60 dư 7, 19, 31, hoặc 43, đảo các số có dạng $3*x^2+y^2$. **Hay số đó chia cho 12 dư 7.**
    - Nếu số đó chia 60 dư 11, 23, 47, hoặc 59, đảo các số có dạng $3*x^2 - y^2$ (Với x>y) **Hay số đó chia cho 12 dư 11.**
    - Còn lại, không làm gì cả.
4) Bắt đầu từ số nhỏ nhất trong sàng.
5) Lấy các số tiếp theo trong sàng được đánh dấu là prime.
6) Thêm vào danh sách kết quả.
7) Bình phương số đó và đánh dấu các bội số của bình phương số đó là không phải số nguyên tố.
8) Lặp lại bước 5 cho tới bước 8.

Code C++:

```
#include <bits/stdc++.h>
using namespace std;
bool sieve[10000001];
void SieveOfAtkin(long limit)
{
//Đánh dấu các số có dạng phương trình bậc 2 có thể là nguyên tố,
//Đây là các giá trị còn lại khi phân tích bằng phương pháp wheel
//factorization.
    for (int x = 1; x * x < limit; x++) {
        for (int y = 1; y * y < limit; y++) {
            int n = (4 * x * x) + (y * y);
            if (n <= limit && (n % 12 == 1 || n % 12 == 5))
                sieve[n] ^= true;
             n = (3 * x * x) + (y * y);
            if (n <= limit && n % 12 == 7)
                sieve[n] ^= true;
             n = (3 * x * x) - (y * y);
            if (x > y && n <= limit && n % 12 == 11)
                sieve[n] ^= true;
        }
    }
    //Loại bỏ tất cả các bội số của bình phương các số nguyên tố
    for (long r = 5; r * r < limit; r++) {
        if (sieve[r]) {
            for (int i = r * r; i < limit; i += r * r)
                sieve[i] = false;
        }
    }
    long count =2;
    for (int a = 5; a < limit; a++)
        if (sieve[a])
            //cout << a << " ";
            count++;
    cout<<count<<endl;
}
int main(void)
{
    long limit = 10000000;
    SieveOfAtkin(limit);
    return 0;
}
```

Độ phức tạp: O(n) [4]

[4] https://en.wikipedia.org/wiki/Sieve_of_Atkin

2.4 Sàng Sundaram

Đây là một thuật toán đơn giản và nhanh giải quyết bài toán tìm các số nguyên tố thỏa một giới hạn nguyên nào đó. Nó được khám phá bởi nhà toán học người Ấn Độ S.P. Sundaram năm 1934.

Ý tưởng thuật toán:

- Bắt đầu với dãy số từ 1 đến n. Từ dãy số này, ta loại bỏ tất cả các số có dạng i + j + 2ij trong đó:
  - i, j ∈ N, 1 ≤ i ≤ j
  - i + j + 2ij ≤ n
- Tất cả các số k còn lại trong dãy số sẽ được tính thành 2k + 1, sau khi tính ta được một danh sách các số nguyên tố lẻ (trừ số 2) nhỏ hơn 2n + 2.
- Ý tưởng này xuất phát từ việc các số nguyên tố, trừ số 2 ra, thì đều là số lẻ. Bên cạnh đó các số có dạng i + j + 2ij, khi được tính thành 2(i + j + 2ij) + 1 sẽ tạo thành một hợp số chứ không phải số nguyên tố. Do đó ta phải loại bỏ các số này đi. Cụ thể ta có:
  Giả sử q là một số nguyên lẻ có dạng 2k + 1, số k sẽ bị loại bỏ khi k có dạng i + j + 2ij. Vì khi đó:

$$q = 2(i + j + 2ij) + 1 = 2i + 2j + 4ij + 1 = (2i + 1)(2j + 1)$$

➔ Rõ ràng q khi này là một hợp số.

Code C++:

```
#include <bits/stdc++.h>
using namespace std;
bool marked[5000001];
void SieveOfSundaram(long n)
{    //Giảm n xuống 1 nửa, vì các số nguyên tố thuộc nửa còn lại sẽ
được tạo ra sau đó khi áp dụng công thức 2k+1.
    long nNew = (n-2)/2;
    //Loại bỏ các số có dạng i+j+2ij
    for (long long i=1; i<=nNew; i++)
        for (long long j=i; (i + j + 2*i*j) <= nNew; j++)
            marked[i + j + 2*i*j] = true;
    //if (n > 2) cout << 2 << " ";
    long count =1;
    for (long i=1; i<=nNew; i++)
        if (marked[i] == false)
            //cout << 2*i + 1 << " ";
            count++;
    cout<<count<<endl;
}
int main()
{    long n = 10000002;
     SieveOfSundaram(n);
     return 0;
}
```

Một cách cài đặt khác cũng khá dễ

Xuất phát từ nhận xét các hợp số có dạng i+j +2ij là tích của 2 số lẻ như phân tích ở trên. Do đó ta chỉ xét các số lẻ và đánh dấu tích của chúng không phải là nguyên tố. Khi thực nghiệm thì đoạn code 1 chạy nhanh hơn 1 chút, nhưng không đáng kể.

Code tham khảo:

```
#include <bits/stdc++.h>
using namespace std;
bool marked[100000000];
void SieveOfSundaram2(long n)
{
    for (long long i=3; i*i<=n; i+=2)
        for (long long j=i; i*j <= n; j+=2)
            marked[i*j] = true;
    /*if (n > 2)
        cout << 2 << " ";*/
     long count =1;
    for (long i=3; i<=n; i+=2)
        if (marked[i] == false)
            //cout << 2*i + 1 << " ";
            count++;
    cout<<count<<endl;
}
int main()
{
     long n = 100000000;
     SieveOfSundaram2(n);
     getchar();
     return 0;
}
```

Độ phức tạp thuật toán ở cả 2 cách cài: O(nlogn) [5]

[5] https://en.wikipedia.org/wiki/Talk%3ASieve_of_Sundaram

2.5 Tổng kết và so sánh hiệu năng

Thực nghiệm được thực hiện trên máy tính có cấu hình: Intel Core 2 (3.0 GHz), RAM 8GB, Windows 64 bit. Cho kết quả như sau:

N	Số lượng SNT	Brute Force	Eratosthenes	Atkin	Sundaram	Sundaram2
Độ phức tạp		O((n√n)/4)	O(nloglogn)	O(n)	O(nlogn)	O(nlogn)
1000	168	0.002	0.001	0.001	0.078	0.001
10000	1229	0.003	0.001	0.002	0.079	0.001
100000	9592	0.01	0.004	0.004	0.077	0.001
1000000	78498	0.124	0.013	0.022	0.089	0.01
10000000	664579	3.109	0.205	0.253	0.211	0.235
100000000	5761455	86.715	2.915	3.695	3.875	4.035
1000000000	50847534	TLE	32.924	43.3	54.677	53.063

Nhìn vào bảng thống kê ta thấy:

- Với $n \leq 10^6$, chỉ cần thuật toán vét cạn thông thường ta có thể giải quyết tốt bài toán đã nêu.
- Với $n = 10^7$, tất cả các thuật toán sàng nguyên tố đều làm tốt
- Với $n = 10^8$, 10^9 tất cả các thuật toán đều không thực hiện tốt. Trong số đó, Sàng Eratosthenes là thuật toán cho kết quả tốt nhất, xấp xỉ 3s.

➔ Cần có những cải tiến để đạt được kết quả tốt hơn.

2.6 Cải tiến Sàng Atkin

Thuật toán đã được trình bày ở mục 3.3. Song ta có thể cải tiến code như sau:
Nhận xét:
- Ở 2 vòng lặp for ban đầu ta phải xét tất cả các cặp x,y (với x*x < limit và y*y < limit). Do đó có một số cặp x, y không phù hợp với các giá trị có dạng phương trình bậc hai.
- Ở vòng lặp cuối khi đếm các số nguyên tố, ta phải xét hết tất cả các số trong đoạn.

Cải tiến:
- Ứng với mỗi dạng phương trình bậc hai, ta sẽ xét các cặp (x,y) với các điều kiện nhất định. Cụ thể:

```
for n ≤ limit, n ← 4x²+y² trong đó x ∈ {1,2,...} và y ∈ {1,3,...} //
xét tất cả giá trị x (chẵn, lẻ) và y lẻ
    if n mod 12 ∈ {1, 5}:
        is_prime(n) ← ¬is_prime(n)   // đảo đánh dấu

for n ≤ limit, n ← 3x²+y² trong đó x ∈ {1,3,...} và y ∈ {2,4,...} //
Chỉ xét các giá trị x lẻ và y chẵn.
    if n mod 12 =7:
        is_prime(n) ← ¬is_prime(n)   // đảo đánh dấu

for n ≤ limit, n ← 3x²-y² trong đó x ∈ {2,3,...} và y ∈ {x-1,x-3,...,1}
// Xét tất cả các cặp (chẵn, lẻ), (lẻ/chẵn) và x>y
    if n mod 12 = 11:
        is_prime(n) ← ¬is_prime(n)   // đảo đánh dấu
```

- Khi loại bỏ các bội số của các số nguyên tố, ta sẽ loại bỏ từ 7 do đã xét chính xác các cặp (x,y) cần thiết ở bước trên.
- Khi đếm các số nguyên tố, ta chỉ đếm đúng các số còn lại trên bánh xe phân tích mà thôi, không xét hết các số trong đoạn. Cụ thể:

```
output 2, 3, 5
for 7 ≤ n ≤ limit, n ← 60 × w + x where w ∈ {0,1,...}, x ∈ s:
    if is_prime(n): output n
```

Code tham khảo:

```
#include <bits/stdc++.h>
using namespace std;
#define boost std::ios::sync_with_stdio(false);
long limit = 100000000;
bool sieve[100000000];
void SieveOfAtkin(long limit)
{

    for (int x = 1; x * x <= limit; x++)
        for (int y = 1; y * y <= limit; y+=2) {
            int n = (4 * x * x) + (y * y);
            if (n <= limit && (n % 12 == 1 || n % 12 == 5))
                sieve[n] ^= true;
    }
```

```
    for (int x = 1; x * x <= limit; x+=2)
        for (int y = 2; y * y <= limit; y+=2) {
            int n = (3 * x * x) + (y * y);
            if (n <= limit && n % 12 == 7)
                sieve[n] ^= true;
    }
    for (int x = 2; x * x <= limit; x++)
        for (int y = x-1; y >=1; y-=2) {
            int n = (3 * x * x) - (y * y);
            if (n <= limit && n % 12 == 11)
                sieve[n] ^= true;

    }
    //danh dau boi cua binh phuong cac so nguyen to
    for (long r = 7; r * r < limit; r++) {
        if (sieve[r]) {
            for (int i = r * r; i < limit; i += r * r)
                sieve[i] = false;
        }
    }
    long count=3;//2, 3, 5 la cac so nguyen to
    //so du quan trong khi chia so n cho 60
    int s[16]={1,7,11,13,17,19,23,29,31,37,41,43,47,49,53,59};

    for(long w=0;w*60 <= limit; ++w)
        for (int x=0;x<=15; ++x)
        {
            long n=60*w + s[x];
            if (n>=7 && n<=limit && sieve[n]) count++;
        }
    cout<<count<<endl;
}
int main(void)
{
    SieveOfAtkin(limit);
    getchar();
    return 0;
}
```

Ngoài ra, các bạn có thể tham khảo các phương pháp cải tiến sàng Atkin khác theo link sau: http://compoasso.free.fr/primelistweb/page/prime/atkin_en.php

Tùy theo tình hình của đội tuyển mà ta có thể triển khai các cách cài đặt phức tạp khác nhau.

2.7 Cải tiến Sàng Eratosthenes

2.7.1 Cải tiến 1

Ta thấy rằng ngoại trừ 2, tất cả các số chẵn đều không phải là số nguyên tố. Vì vậy ta sẽ không xét đến các số chẵn trong thuật toán, khi đó không gian lưu trữ sẽ giảm xuống còn n/2. Điều này sẽ làm tăng tốc độ xử lý và rút ngắn thời gian thực hiện thuật toán.
Ta có thể code như sau:

```
bool * Prime = new bool [n/2];
int eratosthene(){
     memset(Prime,true,n/2);
     //cout<< 2 << "\n";
     long count = 1;
     for (long long i = 3; i < n; i += 2) {
     if (Prime [i / 2]) {
        //cout<< i << "\n";
        count++;
        for (long long j = i * i; j < n; j += 2 * i)
            Prime [j / 2] = false;
     }
     }
     cout<<count<<endl;
     return 0;
}
```

2.7.2 Cải tiến 2

Ta có thể cải tiến thêm bằng cách giảm số lượng các số cần xét xuống nữa. Ta nhận thấy không chỉ các số chẵn, mà các số là bội của 2 hoặc 3 đều không phải là nguyên tố. Do đó, ta tiến hành loại bỏ các số là bội của 2 hoặc 3 ra khỏi danh sách xét.
Các số không phải là bội của 2 hoặc 3 sẽ có bước nhảy lần lượt là +2 và +4 bắt đầu từ 5. Cụ thể ta có: 5 (+2) 7 (+4) 11 (+2) 13 (+4) 17 ... Đây là các số có khả năng là số nguyên tố.
Ta sẽ giảm không gian lưu trữ từ n/2 xuống còn n/3. Dẫn đến tốc độ thuật toán tăng lên. Ta có thể sửa code lại như sau:

Code:

```
bool * Prime = new bool [n/3];
int eratosthene(){
     memset(Prime,true,n/3);
     //cout<< 2 << "\n";
     long count = 2;
     for (long long i = 5, t = 2; i< n; i += t, t = 6 - t) {
     if (Prime [i / 3]) {
        //cout<< i << "\n";
        count++;
        for (long long j = i * i,v = t; j < n; j += v * i, v = 6 -
v)
            Prime [j / 3] = false;
    }
}
     cout<<count<<endl;
     return 0;
}
```

2.7.3 Cải tiến 3

Xét code C++, ta có nhận xét sau:

- Với n = 10000, ta phải sử dụng một mảng có kích thước 40000 bytes (4 bytes hay 32 bits cho một giá trị kiểu int).

- Thay vì sử dụng 32 bits để đánh dấu một số nguyên là nguyên tố hay hợp số, tại sao ta không sử dụng 1 bit? Điều này là hoàn toàn có thể.

- Ta sẽ sử dụng phần tử prime[0] để lưu trữ các số từ 1 đến 32, phần tử prime[1] để lưu trữ các số từ 33 đến 64, và cứ thế tiếp tục…

Prime[0]	Prime[1]
01010000010001010001010001010110	00010100000100000100010100010000
32 31 ... 2 1	64 63 ... 34 33

→ Điều này sẽ làm giảm bộ nhớ, nhưng vẫn chưa cải thiện được thời gian thực hiện.

- Ta tiếp tục cải tiến. Ta nhận thấy các số chẵn đều là hợp số. Vì thế ta sẽ không lưu trữ số chẵn, mà chỉ lưu trữ các số lẻ mà thôi. Ta sẽ dùng prime[0] để lưu trữ các số 1,3,5,…, 63; prime[1] để lưu trữ các số 65, 67, 69,…, 127; và tương tự… Phương pháp này giúp ta tiết kiệm được nhiều bộ nhớ hơn và tốc độ xử lý cũng tăng lên. Cụ thể, lượng bộ nhớ sử dụng cho 1000 số nguyên là 4000/64 = 62.5 bytes.

Prime[0]	Prime[1]
01100100101101001100101101101110	10000001011011010001001010011010
63 61 ... 3 1	127 125 ... 67 65

- Sàng Eratosthenes sau khi tối ưu:

```
#include <bits/stdc++.h>
using namespace std;
#define MAX 100000000
#define SQ 10000
//code se duoc giai thich ben duoi
#define check(n) (prime[n>>6]&(1<<((n&63)>>1)))
#define set(n) prime[n>>6]|=(1<<((n&63)>>1))
int prime[MAX>>6];
int eratosthene(){
     for(int i=3;i<=SQ;i+=2){
          if (!check(i)){
               int tmp = 2*i;
               for(int j=i*i;j<=MAX;j+=tmp){
                    set(j);}
          }
     } return 0;
}
int main(){
     eratosthene();
     return 0;
}
```

Giải thích code:

- Để tìm được phần tử lưu trữ giá trị n trong mảng prime, do mỗi phần tử chứa một đoạn giá trị là 64 , ta thực hiện phép chia n/64 hay n>>6 (dịch sang phải 6 bits). Nghĩa là phần tử thứ prime[n>>6] chứa giá trị của n.
- Để biết được bit nào chứa số n, do mỗi phần tử chứa một khoảng giá trị là 64, ứng với 32 bits, nên bit cần tìm là số dư khi chia n cho 64 (n%64) chia cho 2 (do không tính số chẵn). Để tăng tốc độ ta sử dụng phép toán trên bit thay cho phép toán %. Ta có (n%64)/2 = (n&63)>>1.
- VD: n = 67, ta có n>>6 = 1, và (n&63)>>1 = 1. Nghĩa là phần tử prime[1] lưu giá trị n tại bit thứ 1.

2.8 Kết quả sau cải tiến

Thực nghiệm được thực hiện trên máy tính có cấu hình: Intel Core 2 (3.0 GHz), RAM 8GB, Windows 64 bit. Cho kết quả như sau:

N	Số lượng SNT	Brute Force	Eratosthenes	Atkin	Sundaram	Sundaram2
Độ phức tạp		O((n√n)/4)	O(nloglogn)	O(n)	O(nlogn)	O(nlogn)
1000	168	0.002	0.001	0.001	0.078	0.001
10000	1229	0.003	0.001	0.002	0.079	0.001
100000	9592	0.01	0.004	0.004	0.077	0.001
1000000	78498	0.124	0.013	0.022	0.089	0.01
10000000	664579	3.109	0.205	0.253	0.211	0.235
100000000	5761455	86.715	2.915	3.695	3.875	4.035
1000000000	50847534	TLE	32.924	43.3	54.677	53.063

N	Số lượng SNT	Atkin (cải tiến)	Eratosthenes (cải tiến 1)	Eratosthenes (cải tiến2)	Eratosthenes (cải tiến 3)
1000	168	0	0.001	0	0.001
10000	1229	0	0.002	0.002	0.002
100000	9592	0	0.002	0.002	0.002
1000000	78498	0.015	0.008	0.007	0.006
10000000	664579	0.156	0.064	0.053	0.061
100000000	5761455	2.203	1.491	1.031	0.668
1000000000	50847534	27.161	17.118	11.731	8.860

Nhìn vào bảng thống kê ta thấy sau khi cải tiến đến lần 3, thuật toán Eratosthenes có thể giải bài toán với giới hạn 10^8 rất tốt. Tuy nhiên, khi n = 10^9 thì thuật toán vẫn còn khá chậm nhưng đã có sự cải tiến đáng kể. Trên các hệ thống chấm online thuật toán sẽ chạy nhanh hơn.

3. Bài tập minh họa:

Thầy/Cô có thể tải bộ Test và code mẫu tại link sau: http://bit.ly/2KcuxG4

Mức dễ:

Bài 1: Factor

Số nguyên tố luôn mang đến cho Tom nhiều điều thích thú. Lần này Thầy giao cho Tom một bài toán tưởng chừng như rất dễ nhưng lại đem đến cho Tom một thử thách mới. Bài toán như sau:

Cho bạn một số nguyên dương T là số test cần xử lý. T dòng tiếp theo là T số nguyên dương M, hãy phân tích M ra thành tích các thừa số nguyên tố.

Đây là một bài toán khá đơn giản. Tuy nhiên, Thầy đã ra cho Minh một giới hạn là T <= 10^5 và M <= 10^7

Em hãy giúp Minh vượt qua thử thách này nhé

Input: từ tệp **FACTOR.INP**

- Dòng đầu tiên chứa số lượng các test **T**
- **T** dòng tiếp theo, mỗi dòng chứa số nguyên dương **M**

Output: ghi ra tệp **FACTOR.OUT**

- Xuất ra T **chuỗi là tích các thừa số nguyên tố** nằm trên **T** dòng trả lời cho **T** test ở trên.

Ràng buộc: giới hạn 1s

Ví dụ:

FACTOR.INP	FACTOR.OUT
2 15 30	3*5 2*3*5

Thuật toán:

Ta có thể phân tích số M ra thừa số nguyên tố bằng thuật toán thông thường có độ phức tạp O($\sqrt{M}$) như sau:

```
void factor(long M){
     for(long i = 2; i*i <= M; ++i){
          while (M % i == 0){
               if (m/i !=1) fprintf(fo, "%ld*",i );
               else fprintf(fo, "%ld",i );
               M /= i;
          }
     }
}
```

Ví dụ:

n = 30, i = 2, ans ={2}

n = 15, i = 2, ans = {2}

n = 15, i = 3, ans = {2, 3}

n = 5, i = 4, ans = {2, 3}

n = 5, i = 5, ans = {2, 3, 5}

n = 1

Tuy nhiên, với thuật toán trên ta chỉ giải quyết được bài toán với M tối đa là 10^4 do T là rất lớn.

Cải tiến:

Ta sẽ sử dụng sàng Eratosthenes để phân tích số M ra thừa số nguyên tố với độ phức tạp là O(logn)

Nhận xét: Tại mỗi bước phân tích ta sẽ tìm số nguyên tố nhỏ nhất mà M chia hết.

Từ nhận xét trên, ta sẽ dùng Sàng Eratosthenes để xác định số nguyên tố nhỏ nhất thỏa nhận xét trên trong thời gian O(1)

```
int min_prime(){
     for (long i = 2; i*i <= n; ++i) {
     if (minprime [i] == 0) {
          minprime [i] =i;
          for (long  j = i * i; j <= n; j += i)
               if (minprime[j] == 0)
                    minprime [j ] = i ;
     }
     }
     return 0;
}
//-------------------------
void factor (long m){
     while (m !=1) {
          if (m/minprime[m] !=1)
               fprintf(fo, "%ld*",minprime[m] );
          else fprintf(fo, "%ld\n", minprime[m] );
          m /= minprime[m];
     }

}
```

Code full:

```
#include <bits/stdc++.h>
using namespace std;
#define n 10000000
#define fi "factor.inp"
#define fo "factor.out"
long * minprime = new long [n+1];
FILE* ffi = freopen(fi, "r", stdin);
FILE* ffo = freopen(fo, "w", stdout);
//------------------------------
int min_prime(){
     for (long long i = 2; i*i <= n; ++i) {
     if (minprime [i] == 0) {
          minprime [i] =i;
          for (long long  j = i * i; j <= n; j += i)
               if (minprime[j] == 0)
                    minprime [j ] = i ;
     }
```

```
     }
     for(long long i = 2; i<= n; ++i)
            if (!minprime[i]) minprime[i] = i;
     return 0;
}
//------------------------------
void factor (long m){
     while (m !=1) {
          if (m/minprime[m] !=1)
               fprintf(ffo,"%ld*",minprime[m] );
          else fprintf(ffo,"%ld\n", minprime[m] );
          m /= minprime[m];
     }

}
//------------------------------
void process(){
     long t,m;
     fscanf(ffi,"%ld", &t);
     min_prime();
     for (int i = 1; i<= t ; ++i){
          fscanf(ffi,"%ld", &m);
          factor(m);
     }
}
//------------------------------

int main(){
     process();
     return 0;
}
```

Độ phức tạp: O(nlog(n))

Bài 2: Chú gấu Tommy và các bạn

Chú gấu Tommy là một chú gấu rất dễ thương. Một ngày nọ chú đến trường và được thầy dạy về những con số nguyên tố. Chú và các bạn vô cùng thích thú và lao vào tìm hiểu chúng. Thế nhưng, càng tìm hiểu sâu chú lại càng gặp phải những bài toán khó về số nguyên tố. Hôm nay thầy giao cho cả lớp một bài toán khó và yêu cầu cả lớp ai làm nhanh nhất sẽ được thầy cho bánh. Vì thế, để có bánh ăn, Tommy phải giải bài toán nhanh nhất có thể. Bài toán như sau:

Cho dãy n số nguyên dương x1, x2, …, xn và m truy vấn, mỗi truy vấn được cho bởi 2 số nguyên li, ri. Cho một hàm f(p) trả về số lượng các số x_k là bội của p. Câu trả lời cho truy vấn li, ri là tổng $\sum_{p \in S(li,ri)} f(p)$, trong đó S(li,ri) là tập các số nguyên tố trong đoạn [li,ri]

Bạn hãy giúp chú gấu Tommy giải bài toán này nhé!

Dữ liệu vào: file **TOMMY.INP**

- Dòng đầu tiên chứa số nguyên n ($1 \le n \le 10^5$)
- Dòng thứ 2 chứa n số nguyên dương x1, x2, …, xn ($2 \le x_i \le 10^7$)
- Dòng thứ 3 chứa số nguyên m ($1 \le m \le 50000$). Mỗi dòng i trong m dòng sau chứa 2 số nguyên ngăn cách bởi 1 dấu cách li, ri ($2 \le l_i \le r_i \le 2.10^9$)

Kết quả ra: file **TOMMY.OUT**

- Gồm m dòng, mỗi dòng 1 số nguyên là câu trả lời cho một truy vấn.

Ví dụ:

TOMMY.INP	TOMMY.OUT
6 5 5 7 10 14 15 3 2 10 3 12 4 4	9 7 0

Thời gian: 1s

Giải thích: 3 truy vấn trong test1

1. Truy vấn 1: l = 2, r = 11. Ta cần tính: f(2) + f(3) + f(5) + f(7) + f(11) = 2 + 1 + 4 + 2 + 0 = 9.
2. Truy vấn 2: l = 3, r = 12. Ta cần tính: f(3) + f(5) + f(7) + f(11) = 1 + 4 + 2 + 0 = 7.
3. Truy vấn 3: l = 4, r = 4 → không có số nguyên tố.

Hướng dẫn:

Cách 1: O(m.n.y) với y lá số lượng số nguyên tố trong đoạn [l,r]

B1) Đọc file và xác định giá trị c[i] là số lần xuất hiện của giá trị i trong dãy số.

B2) Dùng sàng Eratosthenes để xác định các số nguyên tố trong đoạn [1..10^7]

B3) Với mỗi truy vấn trong m truy vấn, ta lần lượt xét từng số nguyên tố i trong đoạn [li,ri].

- Với mỗi số nguyên tố i, ta duyệt lại mảng x và đếm số lượng bội của i là f(i);
- Tổng các f(i) chính là kết quả cần tìm.

Cách 2: O(max(m,n).y)

B1) Tương tự cách 1

B2) Dùng sàng số nguyên tố, kết hợp tính các giá trị f(i) (với i là số nguyên tố trong đoạn [1, 10^7)

- Tính f(2)? f(2) = c[2]+ c[4] + c[6] + c[8],...
- Tính f(5)? f(5) = c[5] + c[10]+ c[15] + c[20],...
- Tính f(n)? f(n) = $c[n] + c[2 \cdot n] + c[3 \cdot n] + c[4 \cdot n]$,...

Ta thấy tư tưởng này rất giống với thuật toán sàng Eratosthenes. Vì vậy ta có thể dùng thuật toán sàng và chỉnh sửa lại. Kết quả tính sẽ được lưu trữ vào f[n]

B3) Với mỗi truy vấn trong m truy vấn, ta lần lượt xét từng số nguyên tố i trong đoạn [li,ri]. Ta tính tổng các f(i) chính là kết quả cần tìm.

Cách 3: O(max(m,n))

Để giải bài toán này ta cần giải quyết một số vấn đề sau:

B1, 2) Tương tự cách 2

B3) Bây giờ ta cần tính tổng tiền tố S của mảng num. Với S[i] = f[1] + f[2] + …+f[i]

B4) Sau khi tính tổng tiền tố xong, ta có thể tính toán tổng số lượng phần tử giữa l và r trong thời gian O(1), nghĩa là ta tính s[r] – s[l-1]. Bây giờ ta có thể đọc các truy vấn và trả lời chúng dễ dàng.

Cần lưu ý là cận phải r có thể lớn hơn 10^7, vì vậy ta có thể giải r xuống chỉ còn 10^7 thôi và tất cả các số được cho đều bé hơn hoặc bằng 10^7.

Code

```
#include <bits/stdc++.h>
using namespace std;
#define MAX 10000001
int prime[MAX];
long c[MAX], f[MAX];
long long sum[MAX];
long t,n,m;
//FILE * fi = freopen("tommy.inp","r",stdin);
//FILE * fo = freopen("tommy.out", "w", stdout);
ifstream fi ("tommy.inp");
ofstream fo ("tommy.out");

//---------------------------
int eratosthene(){
     for(long long i=2;i*i<=MAX; ++i){
          if (!prime[i]){
               for(long long j=i;j<=MAX;j+=i){
                    f[i] += c[j];
                    prime[j]=true;
               }
          }
     }
     return 0;
}
//-------------------------------
void process(){
     fi>>n;
     for(long i = 1; i<= n; ++i){
```

```
          fi>>t;
          ++c[t];
     }
     eratosthene();
     for(long i = 2; i<= MAX ; ++i)
          sum[i] = sum[i-1] + f[i];

     fi>>m;
     long li,ri;
     for (long i = 1; i<= m; ++i){
          fi>>li>>ri;
          if (ri>1E7) ri = 1E7;
          fo<<sum[ri] - sum[li-1]<<'\n';
     }
}
//--------------------------------
int main(){
     ios_base::sync_with_stdio(false);
     process();
     return 0;
}
```

Bài 3: Hoán đổi

Trong giờ giải lao, do lớp vừa học xong các kiến thức về số nguyên tố, nên lớp trưởng của John đã suy nghĩ ra một trò chơi về dãy số cũng khá thú vị. Trò chơi như sau:

Cho một dãy số $a[1], a[2], ..., a[n]$, gồm các số nguyên phân biệt từ 1 đến n. Nhiệm vụ là ta phải sắp xếp các số theo thứ tự tăng dần theo qui tắc sau (có thể áp dụng nhiều lần):

1. Chọn trong dãy số 2 chỉ số i, j ($1 \le i < j \le n$; $(j - i + 1)$ là số nguyên tố)
2. Hoán đổi 2 số tại vị trí i, j.

Không cần thiết phải sử dụng số lần nhỏ nhất các qui tắc trên, nhưng không được sử dụng vượt quá 5*n lần.

Input: vào từ file **SWAP.INP** như sau:

- Dòng đầu tiên chứa số nguyên n ($1 \le n \le 10^5$)
- Dòng tiếp theo chứa n số nguyên phân biệt a[1], a[2], ..., a[n] ($1 \le a[i] \le n$).

Output: ghi ra file **SWAP.OUT** như sau:

- Dòng đầu tiên, in số nguyên k ($0 \le k \le 5n$) là số lần qui tắc được sử dụng.
- Dòng tiếp theo in k cặp (i,j) đã hoán đổi. Với i,j thỏa yêu cầu đề bài.

Nếu có nhiều đáp án ta in một đáp án bất kỳ.

Ví dụ:

SWAP.INP	SWAP.OUT
3 3 2 1	1 1 3
2 1 2	0
4 4 2 3 1	3 2 4 1 2 2 4

Thuật toán:O(n+m) với m là số lần hoán đổi

Ý tưởng duyệt: với mỗi số i chưa đúng vị trí, gọi y[i] là vị trí hiện tại của số i. Ta tìm vị trí t phù hợp lần lượt từ vị trí thứ i, i+1, i+2… Khi tìm được t ta hoán đổi 2 giá trị tại y[i] và t, ghi nhận vị trí mới. làm tương tự cho đến khi tất cả các số đều đúng vị trí.

Nhận xét: với mỗi số i ta đã tìm vị trí xa nhất thỏa yêu cầu để hoán đổi nên có thể thấy đây là thuật toán tốt. Thực tế cài đặt cho thấy số lần hoán đổi thỏa yêu cầu đề bài.

Chi tiết thuật toán:

1. Khi đọc dãy số ta tiến hành lưu lại vị trí của từng số a[i] ban đầu là y[a[i]] = i
2. Xét từng số i = 1, 2, 3, …, n. Với mỗi số ta thực hiện nhiều lần các bước sau:

a. Gọi j = y[i] là vị trí hiện tại của số i trong dãy số. Trong khi j>i (i chưa đúng chỗ), ta thực hiện tìm vị trí t thích hợp để hoán đổi giá trị i với t = i, i+1, i+2, …

b. Khi tìm được vị trí t phù hợp, ta thực hiện đếm số lần hoán đổi và cập nhật vị trí như sau

- y[a[t]] = j
- y[a[j]] = t

c. Hoán đổi a[t] và a[j]

Code tham khảo:

```
#include<bits/stdc++.h>
using namespace std;
#define check(n) (prime[n>>6]&(1<<((n&63)>>1)))
#define set(n) prime[n>>6]|=(1<<((n&63)>>1))
#define MAX 100000000
long y[100000005],c=0,a[100000005],n,t,j;
vector< pair < long , long > > v;
int prime[MAX>>6];
//--------------------
void eratos(){
        for (long i=3;i*i<=MAX;i+=2)
        if (!check(i)){
            long t=2*i;
            for (long j = i*i; j<= MAX; j+=t)
                set(j);
        }
}
//---------------------
bool checkprime(long m){

    if ((m==1)||((m>2)&&(!(m%2)))||((m%2)&&check(m))) return false;
    return true;
}
void process(){
    FILE * fi = freopen("swap.inp","r", stdin);
    FILE * fo = freopen("swap.out","w", stdout);
    scanf("%d",&n);
    eratos();
    for (long i =1; i<=n; i++){
        scanf("%d",&a[i]);
        y[a[i]] = i;
    }

    for (long i=1;i<=n;i++)
    {
        for (j=y[i];j>i;)
        {
            t=i;
            while (!checkprime(j-t+1))
                t++;
            c++;
            y[a[t]]=j;
            y[a[j]]=t;
            v.push_back(make_pair(t,j));
            swap(a[t],a[j]);
            j=t;
        }
```

```
    }
    cout<<c<<'\n';
    for (long i=0;i<c;i++)
        cout<<v[i].first<<" "<<v[i].second<<'\n';
}
int main()
{
    ios_base::sync_with_stdio(false);
    process();
    return 0;
}
```

Mức khá

Bài 4: SumNT

Nhận thấy Tom là một học sinh xuất sắc và bị hấp dẫn rất nhiều về số nguyên tố, Thầy giáo lại quyết định cho Tom một thử thách tiếp theo là tìm tổng của N số nguyên tố đầu tiên. Do giới hạn khá lớn nên Tom hơi bị lúng túng. Em hãy giúp anh ấy tìm cách giải bài toán này thật nhanh.

Input: file SUMNT.INP

- Dòng đầu tiên chứa số lượng các test **T**
- **T** dòng tiếp theo, mỗi dòng chưa số nguyên dương M

Output: file SUMNT.OUT

- Xuất ra **T** số nằm trên **T** dòng trả lời cho **T** test ở trên.

Ràng buộc:

$1 <= T <= 80$

$1 <= N <= 10^6$

Ví dụ:

SUMNT.INP	SUMNT.OUT
2 6 16	41 160

Chú ý: Số nguyên tố có thể lên đến 10^8

Giải thích

Ta có T = 2

Khi N = 6, tổng các số nguyên tố là = 2+3+5+7+11+13= 28

Khi N = 11, tổng các số nguyên tố là = 2+3+5+7+11+13+17+19+23+29+31= 160

Thuật toán: O(nlogn/2)

- Do $N = 10^6$, khi đó số nguyên tố có độ lớn lên đến 10^8 và giới hạn thời gian cho mỗi test là 1s, nên các thuật toán chưa cải tiến đều không đáp ứng được nhu cầu bài toán.
- Ta sẽ sử dụng thuật toán Sàng Eratosthenes cải tiến 3 là lựa chọn tốt nhất.

Code tham khảo:

```
#include <bits/stdc++.h>
using namespace std;
#define n 100000000
#define check(n) (prime[n>>6]&(1<<((n&63)>>1)))
#define set(n) prime[n>>6]|=(1<<((n&63)>>1))
long long sum[10000001];
int prime[n>>6];
long t,m;
//--------------------------
void process(){
     FILE* fi = freopen("sumnt.inp","r",stdin);
     FILE* fo = freopen("sumnt.out","w",stdout);
     scanf("%ld", &t);
     for (long long i =1; i<=t; ++i){
          scanf("%ld", &m); getchar();
```

```
          printf("%lld\n", sum[m]);
     }
}
//----------------------------
int eratosthene(){
     for(long long i=3;i*i<=n;i+=2){
          if (!check(i)){
               long long tmp = 2*i;
               for(long long j=i*i;j<=n;j+=tmp){
                    set(j);
               }
          }
     }
     sum[1] = 2;
     long long cs = 1;
     for(int i=3;i<=n; i+=2)
          if (!check(i))
          {    cs++;
               sum[cs] = sum[cs -1] + i;
          }
     return 0;
}
//----------------------
int main(){
     eratosthene();
     process();
     return 0;
}
```

Bài 5: Thuyền trưởng

Thuyền trưởng Prime đang đi thám hiểm đến vùng đất bí ẩn giữa đại dương mênh mông cùng với quân đoàn tinh nhuệ nhất của ông ta. Trên đường đi có rất nhiều thế lực đen tối tấn công vào tinh thần của các binh sĩ. Chúng làm cho binh sĩ hoãn loạn không làm chủ được bản thân. Vì thế, ông đã quyết định ném một số binh sĩ xuống biển. Các binh sĩ có bị ném xuống biển hay không tùy thuộc vào số hiệu họ mang trên người.

Con tàu được chia thành 3 phần: LEFT, RIGHT và CENTRAL. Mỗi binh sĩ trên tàu được gắn một số hiệu nhận dạng (id). Và theo số id đó họ sẽ làm việc trên một phần của con tàu. Khu vực làm việc được qui định như sau đối với một binh sĩ: Các binh sĩ được sắp làm việc phải có số id là số nguyên tố và không chứa số 0. Ngoài ra từng khu vực sẽ có qui định riêng đối với binh sĩ như sau:

- Khu vực **CENTRAL**: anh ta sẽ làm việc ở phần giữa của con tàu nếu
  - Khi bỏ dần các chữ số bên trái của id lần lượt theo thứ tự thì số còn lại cũng phải là số nguyên tố.
  - Tương tự cho các số nằm bên phải của số id.

VD: Xét số id = 3137, sẽ làm việc ở khu vực giữa vì ta có các số 3137, {313, 31, 3}, {137, 37, và 7} đều là số nguyên tố.

- Khu vực **LEFT**: anh ta sẽ làm việc ở phần trái của con tàu nếu
  - Khi bỏ dần các chữ số bên trái của id lần lượt theo thứ tự thì số còn lại cũng phải là số nguyên tố.

VD: Xét số id = 1367, sẽ làm việc ở khu vực trái vì ta có các số 1367, 367, 67, và 7 là các số nguyên tố.

- Khu vực **RIGHT**: anh ta sẽ làm việc ở phần trái của con tàu nếu
  - Khi bỏ dần các chữ số bên phải của id lần lượt theo thứ tự thì số còn lại cũng phải là số nguyên tố.

VD: Xét số id = 2333, sẽ làm việc ở khu vực phải vì ta có các số 2333, 233, 23, và 2 là các số nguyên tố.

DEAD: Binh sĩ bị ném xuống sông là binh sĩ không làm việc ở bất cứ phần nào của con tàu.

Input: từ file **CAPTAIN.INP** có dạng

- Dòng đầu tiên chứa số nguyên T, là số binh sĩ trên con tàu.
- T dòng tiếp theo chứa số id của họ.

Output: ghi ra file **CAPTAIN.INP** có dạng:

- In các giá trị LEFT, RIGHT, CENTRAL hay DEAD theo thứ tự.

Ví dụ:

CAPTAIN.INP	CAPTAIN.OUT
5 3137 1367 2333 101 12	CENTRAL LEFT RIGHT DEAD DEAD
4 43 23 66 29	LEFT CENTRAL DEAD RIGHT

Ràng buộc:
1 <= T <= 50
1 <= id <= 10^8

Thuật toán: O(nlogn/2)
- Do số id lớn nên ta sử dụng thuật toán sàng eratosthenes cải tiến 3 để xử lý.
- Với mỗi số id ta thực hiện kiểm tra như trong đề để gán nhãn cho từng id là LEFT, RIGHT, CENTRAL hay DEAD.

Code tham khảo:

```
#include <bits/stdc++.h>
using namespace std;
#define MAX 100000000
#define check(n) (prime[n>>6]&(1<<((n&63)>>1)))
#define set(n) prime[n>>6]|=(1<<((n&63)>>1))
int prime[MAX>>6];
int n;
long a,label[50];

void eratos(){
        for (long i=3;i*i<=MAX;i+=2)
        if (!check(i)){
            long t=2*i;
            for (long j = i*i; j<= MAX; j+=t)
                set(j);
        }
}
bool checkprime(long y){
    if((y==1)||((y>2)&&(!(y%2)))||((y%2)&&check(y))) return false;
    return true;
}
int check_label(long m){
    string s,tmp;
    stringstream stream;
    stream << m;
    s = stream.str();
    int tam = s.find('0');
    if (tam>0) return 4;//neu id chua so 0
    //Bo lan luot cac so ben trai va kiem tra
    bool f1 = true;
    for(int i=1;i<s.length(); ++i){
        tmp = s.substr(i);//tao xau con khi bo i ky tu trai
        long x = atol(tmp.c_str());
        if (!checkprime(x)){
            f1=false;
            break;
```

```
        }
    }
    //Bo lan luot cac so ben phai va kiem tra
    bool f2 = true;
    for(int i = 1; i<s.length(); ++i){
        tmp = s.substr(0,s.length()-i);//tao xau con khi bo i ky tu
phai
        long x = atol(tmp.c_str());
        if (!checkprime(x)){
            f2 = false;
            break;
        }
    }
    if (f1 && f2) return 1;//central
    else if (f1) return 2;//left
        else if (f2) return 3;//right
            else return 4;//dead

}
void process(){
    FILE* fi = freopen("captain.inp","r",stdin);
    FILE* fo = freopen("captain.out","w",stdout);

    eratos();
    scanf("%d",&n);
    for (int i=0; i<n; ++i){
        scanf("%ld", &a);
        if (!checkprime(a)) label[i] = 4;
        else label[i] = check_label(a);
    }
    //in ket qua
    for (int i=0; i<n; ++i){
        if (label[i]==1) printf("%s\n","CENTRAL");
        else if (label[i] == 2) printf("%s\n","LEFT");
             else if(label[i] == 3) printf("%s\n", "RIGHT");
                  else printf("%s\n", "DEAD");
    }
}
int main()
{
    process();
    return 0;
}
```

Mức khó

Bài 6: Prime Not-Prime

Tim là một nhà toán học tài ba, anh đang cố gắng giải một bài toán khó. Nhưng dường như đây là một bài toán quá khó để giải trên giấy. Vì vậy, anh ta đã hỏi bạn một vấn đề là tính tổng của |p-x|, với p là nguyên tố trong đoạn [1, n] và x là số không nguyên tố trong đoạn [1, n]. Câu trả lời có thể rất lớn, vì thế đáp án là phần dư khi chia cho $10^9 + 9$

Input: từ file **PNP.INP** có dạng

- Dòng đầu tiên chứa một số nguyên T, là số lượng test.
- T dòng sau chứa các số nguyên từ 1 đến n.

Output: ghi ra file **PNP.OUT** gồm:

- T dòng, mỗi dòng chứa một số nguyên là tổng tìm được

Ràng buộc

$1 \le T \le 20$

$1 \le n \le 10^8$

Ví dụ:

PNP.INP	PNP.OUT
2 2 5	1 11

Giải thích:

- Các số nguyên tố trong đoạn [1, 2] là 2 và số không nguyên tố là 1. Vì vậy, ta có |2 -1| = 1
- Các số nguyên tố trong đoạn [1, 5] là {2, 3, 5} và các số không nguyên tố là {1, 4}. Vì vậy ta có: |2-1|+|2-4|+ |3-1|+|3-4| + |5-1|+|5-4| = 1+2 + 2+1 + 4+1 = 11

Thuật toán:

Cách 1: O(T.n²)

1) Dùng sàng số nguyên tố để đánh dấu các số nguyên tố.

2) Lần lượt đọc T số n, mới mỗi n ta xét đoạn [1, n]. Với mỗi số nguyên tố i, ta tính tổng sum += |i-j| với j là các số không phải là nguyên tố trong đoạn đang xét.

Cách 2: O(T.n/2)

Chúng ta xét n = 5, chúng ta phải tính |2-1|+|2-4|+|3-1|+|3-4|+|5-1|+|5-4|

|2-1| = |1-2|

|2-4| = |2-3| + |3-4|

|3-1| = |1-2| + |2-3|

|3-4| = |3-4|

|5-1| = |1-2| + |2-3| + |3-4| + |4-5|

|5-4| = |4-5|

- Có bao nhiêu lần |1-2| được tính? 3 lần. Tại sao? Lý do là vì có 1 số không nguyên tố bên trái của đoạn là số 1, và 3 số nguyên tố bên phải của đoạn là 2, 3, 5.
- Có bao nhiêu lần |2-3| được tính? 3 lần. Bởi vì có 1 số không nguyên tố bên trái là 1 và 2 số nguyên tố bên phải là 3, 5 nên đã tạo ra 2 lần tính |2-3|. Bên cạnh đó ta có 1 số nguyên

tố bên trái là 2, và 1 số không nguyên tố bên phải là 4, tiếp tục tạo ra 1 lần tính nữa. Tổng là 3 lần.

- Có bao nhiêu lần |3-4| được tính? 3 lần. Giải thích là tương tự.
- Từ phân tích trên, ta cần chú ý đến số lần |x-(x+1)| được tính vào kết quả. Để làm được điều này, ta cần biết:

- leftprime: Số lượng các số nguyên tố bên phía trái (đoạn từ 1..x),
- rightNotPrime: số lượng các số không nguyên tố bên phía phải (đoạn từ x+1..n)
- leftNotPrime: Số lượng các số không nguyên tố bên phía trái (đoạn từ 1..x),
- rightPrime: số lượng các số nguyên tố bên phía phải (đoạn từ x+1..n)

➔ Số lần |x-(x+1)| được tính vào kết quả là: leftPrime*rightNotPrime + leftNotPrime*rightPrime

- Ta thực hiện lặp với x = 1..n-1 và sau mỗi vòng lặp, ta cập nhật leftPrime, rightPrime,leftNotPrime, rightNotPrime dựa trên việc x+1 là nguyên tố hay không.
- Khởi tạo ban đầu: leftPrime = 0, leftNotPrime = 1, rightPrime =số lượng số nguyên tố trong đoạn 1..n, rightNotPrime = n – rightPrime -1.

Code tham khảo:

```
#include <bits/stdc++.h>
using namespace std;
#define check(n) (prime[n>>6]&(1<<((n&63)>>1)))
#define set(n) prime[n>>6]|=(1<<((n&63)>>1))
#define MAX 100000000
#define MOD 1000000009
typedef long long ll;
int prime[MAX>>6];
//--------------------------
void eratos(){
        for (long i=3;i*i<=MAX;i+=2)
        if (!check(i)){
            long t=2*i;
            for (long j = i*i; j<= MAX; j+=t)
                set(j);
        }
}
//--------------------------
void process(){
    ll test, n;
    ll allPrime;
    ll leftPrime, leftNotPrime;
    ll rightPrime, rightNotPrime;

    FILE* fi = freopen("pnp.inp","r",stdin);
    FILE* fo = freopen("pnp.out","w",stdout);

    scanf("%lld", &test);
    while(test --)
    {
```

```
        scanf("%lld",&n);
        allPrime = 1;
        for(ll i = 3; i <= n; i+=2)
            if(!check(i))
                allPrime ++;

        leftPrime = 0;
        leftNotPrime = 1;
        rightPrime = allPrime;
        rightNotPrime = n-allPrime-1;

        ll i = 1;
        ll res = 0;
        while(i < n)
        {
            res = (res + leftPrime*rightNotPrime) % MOD;
            res = (res + leftNotPrime*rightPrime) % MOD;
            ll tmp = i+1;
            if((tmp == 2) ||(tmp%2)&&(!check(tmp)))//i+1 is prime
            {
                leftPrime ++;
                rightPrime --;
            }
            else
            {
                leftNotPrime ++;
                rightNotPrime --;
            }
            i ++;
        }
        printf("%lld\n", res);
    }
}
int main()
{
    eratos();
    process();
    return 0;
}
```

4. Kết luận

Các thuật toán sàng số nguyên tố rất hay và đa dạng. Các dạng toán áp dụng cũng rất phong phú. Vì thế nếu biết cách vận dụng tốt các thuật toán này sẽ giúp học sinh hoàn chỉnh hơn các kiến thức được học về số nguyên tố.

Vì đối tượng là học sinh chuyên Tin lớp 10, nên Chuyên đề hướng đến trang bị cho các em kiến thức về từng loại sàng số nguyên tố, và so sánh hiệu năng giữa các thuật toán, tiến hành cải tiến các thuật toán sàng Atkin và trọng tâm là sàng Eratosthenes giúp giải quyết được các bài toán có giới hạn lên đến 10^8.

Tuy nhiên, thực tế vẫn có một cách cải tiến khác để đạt được các giới hạn n lớn hơn trong thời gian 1s nhưng cài đặt khá phức tạp như:

- Ta có thể cải tiến tiếp thuật toán Eratosthenes theo đoạn (Segmented sieve of Eratosthenes) rất hay. Tuy nhiên, thuật toán cài đặt phức tạp hơn và dành cho các CPU thế hệ mới, khi mà bộ nhớ đệm (Cache L1) có dung lượng lớn. Kết quả đem lại là rất tốt với N = 10^9. Ta có thể tham bài viết theo cách này tại link sau: https://primesieve.org/segmented_sieve.html
- Bên cạnh đó, việc áp dụng phương pháp Wheel factorization để cải tiến sàng Eratosthenes cũng là một hướng cải tiến khác cho hiệu quả tốt. Thống kê trên wiki: https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes
- Ta cũng có thể kết hợp sàng Eratosthenes cơ bản với thuật toán Meissel – Lehmer sẽ đem lại kết quả khoảng N = 10^9, nếu CPU mạnh có thể đạt đến 10^{10}. Tuy nhiên, cách cài đặt là khá phức tạp và chủ yếu dùng để đếm số lượng số nguyên tố chứ chưa thể liệt được hết các số nguyên tố. Không phù hợp cho các bài toán cần đến giá trị của từng số nguyên tố. Tham khảo các link sau:
  https://en.wikipedia.org/wiki/Meissel%E2%80%93Lehmer_algorithm
  http://mathworld.wolfram.com/LehmersFormula.html
  https://sites.google.com/site/kc97ble/arithmetic/lehmer---dhem-so-luong-so-nguyen-to-nho-hon-n