Mục lục

CHUYÊN ĐỀ I

THUẬT TOÁN VÀ PHÂN TÍCH THUẬT TOÁN

1 Thuật toán

a) Khái niệm

Thuật toán có nghĩa là: dãy các bước có thứ tự chính xác để giải quyết được một bài toán cụ thể, theo đó với mỗi bộ dữ liệu vào giải thuật cho một kết quả.

Để hiểu rõ hơn hãy lấy một ví dụ: Để nấu một món ăn mới, người ta đọc hướng dẫn và thực hiện theo từng bước một, theo trình tự nhất định và kết quả là người ta sẽ có một món ăn hoàn hảo. Tương tự như vậy, các thuật toán thực hiện một nhiệm vụ trong lập trình để có được kết quả như mong đợi.

b) Các đặc điểm của thuật toán

- **Rõ ràng:** Thuật toán phải rõ ràng. Mỗi bước của nó phải rõ ràng về mọi mặt và chỉ dẫn đến một ý nghĩa.
- **Đầu vào được xác định rõ:** Nếu thuật toán yêu cầu nhận đầu vào, thì đó phải là đầu vào được xác định rõ.
- **Đầu ra được xác định rõ:** Thuật toán phải xác định rõ đầu ra nào sẽ được tạo ra và nó cũng phải được xác định rõ.

- **Hữu hạn:** Thuật toán phải hữu hạn, tức là nó không được kết thúc trong một vòng lặp vô hạn.
- **Độc lập với ngôn ngữ:** Thuật toán được thiết kế phải độc lập với ngôn ngữ, tức là nó phải chỉ là các quy trình, câu lệnh có thể được thực hiện bằng bất kỳ ngôn ngữ nào và kết quả đầu ra sẽ giống như mong đợi.
- **Tổng quát:** Thuật toán có thể áp dụng để giải mọi bài toán có dạng đã cho.

2 Phân tích thuật toán

a) Tại sao cần phải phân tích thuật toán ?

Mục tiêu của phân tích thuật toán không chỉ là để so sánh, đánh giá giúp cho việc lựa chọn thuật toán tốt mà còn dựa vào kết quả phân tích đánh giá đó để hiệu chỉnh, cải tiến thuật toán đã có được tốt hơn. Một cách để đánh giá thuật toán là dựa vào số câu lệnh mà chương trình nguồn cần thực hiện trên một tập dữ liệu vào.
Việc chú ý đến các phép toán được thực hiện nhiều lần là điều cần thiết khi cài đặt thuật toán bằng chương trình. Ví dụ, người lập trình cần quan tâm rất lớn để cải tiến các "vòng lặp" vì các phép toán trong đó là các phép toán được thực hiện lặp lại nhiều lần nhất.
Cách đánh giá thời gian thực hiện thuật toán độc lập với hệ thống máy tính dẫn tới khái niệm **độ phức tạp thuật toán**. Thời gian để thực hiện thuật toán phụ thuộc rất nhiều yếu tố. Một yếu tố rất quan trọng là kích thước của dữ liệu vào. Dữ liệu càng lớn thì thời gian thực hiện thuật toán càng lớn.

b) Phân tích thuật toán

Ta gọi $T(n)$ là thời gian thực hiện của một thuật toán đang phân tích.
Giả sử n là một số không âm, $T(n)$ và $f(n)$ là các hàm xác định dương với mọi n.
Ta viết : $T(n) = O(f(n))$ nếu tồn tại các hằng số dương c và n_0 sao cho $T(n) \leq c.f(n)$ với mọi $n > n_0$
Ví dụ với $T(n) = n^2 + 1$ thì $T(n) = O(n^2)$
Thật vậy chọn $c = 2$ và $n_0 = 1$ khi đó với mọi $n \geq 1$ Ta có:$T(n) = 2n^2+1 \leq 2n^2 = 2f(n)$.
Xác định độ phức tạp thuật toán thông qua hàm O:

- **Quy tắc hằng số:** Nếu một thuật toán có thời gian thực hiện $T(n) = O(c_1 f(n))$ với c_1 là một hằng số dương thì có thể coi thuật toán đó có độ phức tạp là O(f(n)).
- **Quy tắc cộng:** Giả sử $T_1(n)$ và $T_2(n)$ là thời gian thực hiện của 2 đoạn chương trình P_1 và P_2 mà $T_1(n) = O(f(n))$ và $T_2(n) = O(g(n))$ thì thời gian thực hiện đoạn P_1 nối tiếp P_2 sẽ là : $T_1(n) + T_2(n) = O(max(f(n), g(n))$.
- **Quy tắc nhân:** Nếu tương ứng với P_1 và P_2 là $T_1(n) = O(f(n)), T_2(n) = O(g(n))$ thì thời gian thực hiện P_1 và P_2 lồng nhau sẽ là : $T_1(n).T_2(n) = O(f(n).g(n))$.

Giả sử rằng, các lệnh gán không chứa các lời gọi hàm. Khi đó để đánh giá thời gian thực hiện một chương trình chúng ta xuất phát từ các lệnh đơn rồi đánh giá các lệnh phức tạp hơn, cuối cùng để đánh giá được thời gian thực hiện của một chương trình, cụ thể:

1. Thời gian thực hiện các lệnh đơn: gán, đọc, viết là $O(1)$.

2. Lệnh hợp thành: thời gian thực hiện lệnh hợp thành được xác định bởi $S_1, S_2, ..., S_m$ tương ứng là $O(f_1(n)), O(f_2(n)), ..., O(f_m(n))$. Khi đó thời gian thực hiện của lệnh if là: $O(max(f_1(n), f_2(n), ..., f_m(n))$.

3. Lệnh if: Giả sử thời gian thực hiện các lệnh S_1, S_2 là $O(f(n))$ và $O(g(n))$ tương ứng. Khi đó thời gian thực hiện lệnh if là $O(max(f(n), g(n)))$.

4. Lệnh *switch*: Lệnh này được đánh giá như lệnh if.

5. Lệnh *while*: Giả sử thời gian thực hiện lệnh S (thân của while) là $O(f(n))$ và $g(n)$ là số lần tối đa thực hiện lệnh S, khi đó thời gian thực hiện lệnh while là $O(f(n).g(n))$.

6. lệnh $do...while$, for tương tự như *while*.

Ví dụ 1:

```
[1]     cin >> n;
[2]     int s1 = 0;
[3]     for(int i=1;i<=n;i++)
[4]          s1 = s1+i;
[5]     int s2 = 0;
[6]     for(int i=1;i<=n;i++)
[7]          s2 = s2 + i*i;
[8]     cout << s1 << endl;
[9]     cout << s2;
```

Thời gian thực hiện của chương trình phụ thuộc vào n. Các lệnh (1), (2), (4), (5), (7), (8), (9) có độ phức tạp là $O(1)$. Lệnh lặp for (3) có số lần lặp là n nên lệnh (3) có độ phức tạp là $O(n)$, tượng tự với lệnh (6).
Ví dụ 2:

```
[1]     int c=0;
[2]     for(int i=1;i<=2*n;i++)
[3]          c=c+1;
[4]     for(int i=1;i<=n;i++)
[5]          for(int j=1;j<=n;j++)
[6]               c=c+1;
```

Thời gian thực hiện của chương trình phụ thuộc vào n. Các lệnh (1), (3), (6)có độ phức tạp là $O(1)$. Lệnh for (2) có số lần lặp là $2n$ nên có độ phức tạp $O(2n)$. Lệnh for (5) có độ phức tạp là $O(n)$, for (5) được lồng trong for (4) nên lệnh for (4) có độ phức tạp $O(n^2)$.

CHUYÊN ĐỀ II

TOÁN TRONG TIN HỌC

1 Hệ đếm

Hệ đếm được hiểu là tập các kí hiệu và quy tắc sử dụng tập các kí hiệu đó để biểu diễn và xác định giá trị các số. Trong hệ cơ số $b(b > 1)$, các kí hiệu được dùng có giá trị tương ứng $0, 1, \ldots, b-1$. Giả sử N có biểu diễn:

$$d_n d_{n-1} d_{n-2} \ldots d_1 d_0, d_{-1} d_{-2} \ldots d_{-m}$$

trong đó $n+1$ số các chữ số bên trái, m là số các chữ số bên phải dấu phân chia phần nguyên và phần phân số của số N và các d_i phải thỏa mãn điều kiện:

$$0 \leq d_i < b(-m \leq i \leq n)$$

Khi đó giá trị của số N được tính theo công thức:

$$N = d_n b^n + d_{n-1} b^{n-1} + ... + d_0 b^0 + d_{-1} b^{-1} + ... + d_{-m} b^{-m} = \sum_{i=-m}^{n} d_i \times b^i$$

Chú ý: Để phân biệt số được biểu diễn ở hệ đếm nào người ta viết cơ số làm chỉ số dưới của số đó. Ví dụ 10_2 là biểu diễn 10 ở hệ đếm 2.

1.1 Các hệ đếm thường dùng

a) Hệ thập phân

Hệ thập phân (Hệ đếm cơ số 10) là hệ dùng số 10 làm cơ số. Đây là hệ đếm được sử dụng rộng rãi nhất trong các nền văn minh thời hiện đại.
Hệ gồm các chữ số 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 tạo nên.
Ví dụ: $33 = (3 \times 10) + 3$; $5432 = (5 \times 1000) + (4 \times 100) + (3 \times 10) + 2$
Cơ số 10. Tức là, mỗi chữ số trong số được nhân với $33_{10} = 33 \times 10^1 + 3 \times 10^0$
Dạng tổng quát:

$$A = a_n a_{n-1} \ldots a_1 a_0, a_{-1} \ldots a_{-m} = \sum_{i=-m}^{n} a_i 10^i$$

Ví dụ biểu diễn số thực: $25.256 = 2 \times 10^1 + 5 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2} + 6 \times 10^{-3}$

b) Hệ nhị phân

Hệ nhị phân (hay hệ đếm cơ số hai hoặc mã nhị phân) là một hệ đếm dùng hai kí tự để biểu đạt một giá trị số, bằng tổng các lũy thừa của 2.
Chỉ dùng hai kí hiệu 0 và 1.
Biểu diễn số nhị phân:

$$A = \sum_{i=-m}^{n} a_i 2^i$$

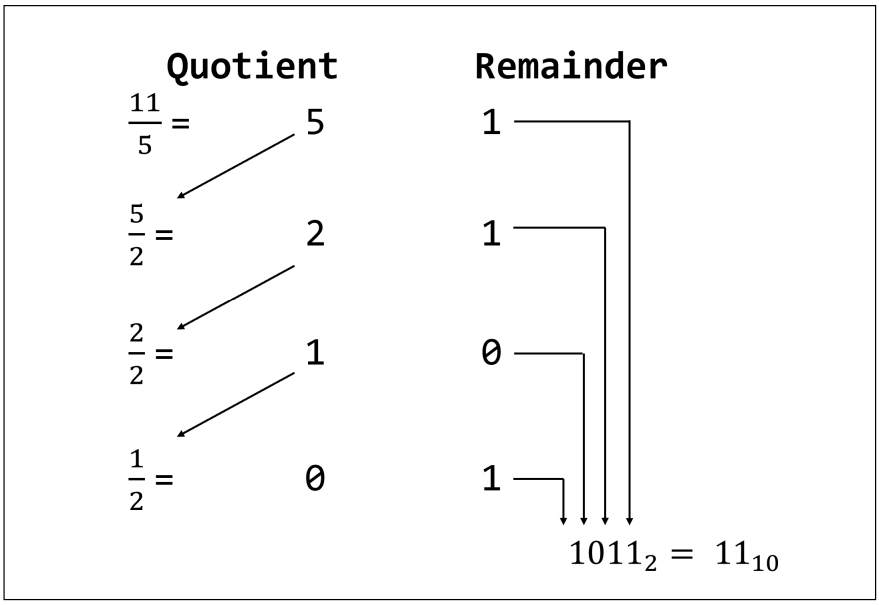
Ví dụ:

- $10_2 = 1 \times 2^1 + 0 \times 2^0 = 2_{10}$
- $101_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5_{10}$
- $100.101 = 1 \times 2^2 + 0 \times 2_1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 4.645^{10}$

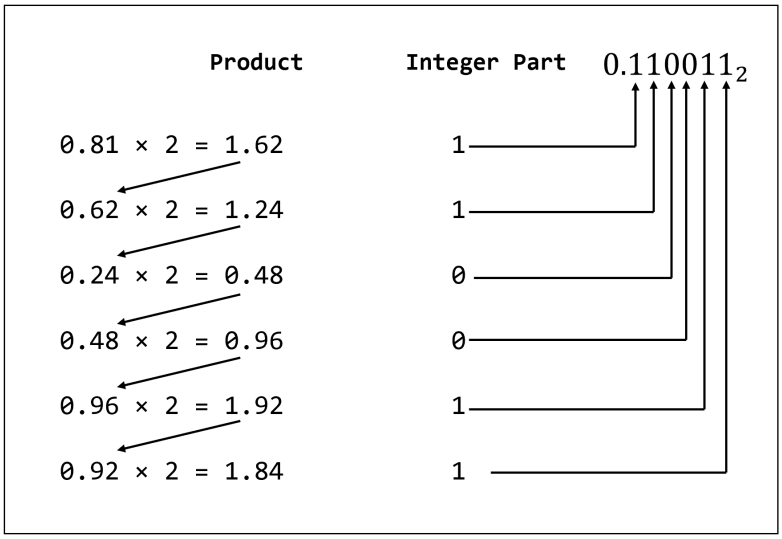
Lưu ý: ở đây các hệ số bằng 0 chúng ta không viết vào cũng được.
Cách chuyển đổi nhị phân sang phần thập phân:
Nhân mỗi chữ số nhị phân với 2^i và cộng vào kết quả.
Cách chuyển đổi từ thập phân sang nhị phân:
Đổi riêng phần nguyên và phần thập phân

- Phần nguyên thập phân sang nhị phân
  - Cách 1:
    - Chia lặp đi lặp lại số đó cho 2. Phép chia dừng lại khi chia lần cuối cùng bằng 0.
    - Lấy các số dư theo chiều đảo ngược sẽ được số nhị phân cần tìm.
  - Cách 2:
    - Phân tích số đó thành tổng của các số 2^i.
- Phần thập phân sang nhị phân
  - Nhân liên tiếp phần phân số của số thập phân với 2.
  - Lần lượt lấy phần nguyên của tích thu được sau mỗi lần nhân là kết quả cần tìm.
  - Lấy phần phân số của tích nhân làm số bị nhân trong bước tiếp theo.

Ví dụ 1: Minh họa cách đổi 11_{10} sang nhị phân bằng cách 1



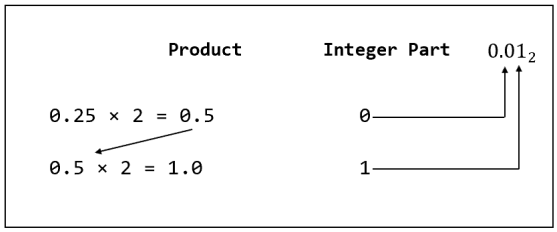
Ví dụ 2: Minh họa đổi 0.81_{10} sang nhị phân



$$0.81_{10} = 0.110011_2$$

Do 0.81 là một số vô tỉ nên ta không thể biết chính xác được phía sau dấu '.' nên ở đây kết quả mình lấy 6 số sau dấu '.'

Ví dụ 3: Minh Họa cách đổi 0.25_{10} sang nhị phân



Do $0.25 = \frac{1}{4}$ là một số hữu tỉ nên theo cách đổi trên ta hoàn toàn có thể xác định được chính xác số chữ số sau dấu '.' và $0.25_{10} = 0.01_2$

Code đổi phần nguyên từ thập phân sang nhị phân bằng đệ quy:

```
[]      DectoBin(int n)
[]      {
```

```
[]            if( n != 0 )
[]            {
[]                 DectoBin(n/2);
[]                 count<<n%2;
[]            }
[]      }
```

c) Hệ thập lục phân

Hệ cơ số mười sáu, hay là hệ hexa, sử dụng các kí hiệu 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, trong đó A, B, C, D, E, F có các giá trị tương ứng 10, 11, 12, 13, 14, 15 trong hệ thập phân.
Lý do sử dụng biểu diễn thập lục phân

- Ngắn gọn hơn ký hiệu nhị phân.
- Trong hầu hết máy tính, dữ liệu nhị phân chiếm theo bội của 4 bit, tương đương với bội của một số thập lục phân.
- Rất dễ dàng chuyển đổi giữa nhị phân và thập lục phân

Cách chuyển từ nhị phân sang thập lục phân:

Cách 1: Đổi từ hệ nhị phân sang thập phân, rồi từ hệ thập phân sang hệ thập lục phân như cách bên trên mình trình bày với hệ nhị phân (muốn từ nhị phân sang thập lục phân ta phải đổi trung gian qua hệ thập phân)

Cách 2: Miêu tả do mỗi chữ số của hệ thập lục phân được biểu diễn bằng 4 bit nhị phân, nên ta thường tính từ dấu '.' nhóm thành 4 bit một rồi chuyển từ nhị phân sang thập lục phân theo 4 bit đó qua cách mình tra bảng dưới đây:

0000 = 0	0100 = 4	1000 = 8	1100 = C
0001 = 1	0101 = 5	1001 = 9	1101 = D
0010 = 2	0110 = 6	1010 = A	1110 = E
0011 = 3	0111 = 7	1011 = B	1111 = F

Ví dụ 1: $10010011_2 = X_{16}$
Theo cách 2 thì mình chia thành 4 bit một từ phải qua trái là 0011 và 1001 ở đây $1001 = 9$ và $0011 = 3 \Rightarrow 10010011_2 = 93_{16}$
Ví dụ 2: $1001111_2 = X_{16}$
Theo cách 2 thì mình chia thành 4 bit một từ phải qua trái là: 1111 và 100, ta thấy ở đây 100 chỉ có 3 bit nên ta phải thêm cho nó 1 bit để đủ 4 bit và chúng ta thêm ở đâu cho đủ? Ở đây ta thêm 1 bit 0 vào bên phải để cho giá trị $0100 = 100$ rồi ta tiếp tục tra bảng. $0100 = 4$ và $1111 = F$ vậy $1001111_2 = 4F16$
Ví dụ 3: $1100.101_2 = X_{16}$
Do ví dụ này mình có thêm dấu '.' vào nên chúng ta phải đổi riêng phần nguyên và phần thập phân và cách đổi tương tự nhiên trên. Ta có phần nguyên là: $1100_2 = C_{16}$ và phần thập phân là 101, khi này ta cần nhớ lại chữ số ngoài cùng bên phải là chữ số ít quan trọng nhất vì vậy khi thêm 1 bit vào cho đủ 4 bit ta thêm bit 0 vào bên phải của 101 tức là $1010 = A$. Vậy $1100.101_2 = C.A_{16}$

Tổng kết lại ở cách đổi này ta cần lưu ý khi đổi phần nguyên ta nhóm 4 bit một từ **phải qua trái** tính từ dấu '.' khi thiếu bit ta **thêm các bit vào bên trái** cho đủ 4 bit rồi tra bảng. Khi đổi phần thập phân ta nhóm 4 bit một nhưng bây giờ ta nhóm **từ trái qua phải** tính từ dấu '.' và khi thiếu bit ta **thêm các bit vào bên phải** cho đủ 4 bit rồi tra bảng.

2 Số nguyên tố

2.1 Định nghĩa

Một số tự nhiên $P(P > 1)$ được gọi là số nguyên tố nếu nó chỉ có đúng 2 ước là 1 là P.

2.2 Kiểm tra tính nguyên tố của 1 số

Để kiểm tra số $n(n > 1)$ có phải là một số nguyên tố hay không ta kiểm tra xem nếu tồn tại một số nguyên $k(2 \leq k \leq n-1)$ mà k là ước của n (n chia hết cho k)thì n không phải là số nguyên tố, ngược lại n là số nguyên tố.
Nếu $n(n > 1)$ không phải là số nguyên tố, ta có thể tách $n = k_1 \times k_2$ mà $2 \leq k_1 \leq k_2 \leq n-1$. Vì $k_1 \times k_1 \leq k_1 \times k_2 = n$ suy ra $k_1 \leq \sqrt{n}$. Do đó việc kiểm tra k từ 2 đến $n-1$ là không cần thiết, ta chỉ cần kiểm tra từ 2 đến $\sqrt{n}$.

```
[]      bool isPrime(int n)
[]      {
[]           if( n < 2 )
[]                return false;
[]           for( int k = 2; k <= sqrt((float)n); k++)
[]                if( n % k == 0)
[]                     return false;
[]
[]           return true;
[]      }
```

Hàm `isPrime(n)` trên kiểm tra lần lượt từng số k trong khoảng $[2, \sqrt{n}]$, để cải tiến ta cần giảm thiểu số lượng các số nguyên cần kiểm tra. Ta có nhận xét, để kiểm tra 1 số nguyên dương n có là số nguyên tố không, ta kiểm tra xem có tồn tại một số nguyên tố $k(2 \leq k \leq \sqrt{n})$ mà k là ước của n thì n không phải là số nguyên tố, ngược lại n là số nguyên tố.
Nếu làm như vậy ta sẽ dẫn tới một vấn đề là:*"kiểm tra xem k có phải số nguyên tố hay không?"* như vậy chúng ta lại có một bài toán con chính là bài toán mà chúng ta đang giải, việc làm này sẽ tăng độ phức tạp của thuật toán và số phép tính chúng ta cần thực hiện.
Thay vì kiểm tra các số nguyên tố ta sẽ kiểm tra các số có tính chất giống với tính chất của số nguyên tố, có thể sử dụng một trong hai tính chất đơn giản sau:

1. Trừ số 2 và các số nguyên tố là số lẻ.
2. Trừ số 2, số 3 các số nguyên tố có dạng $6k \pm 1$ (vì số có dạng $6k \pm 2$ thì chia hết cho 2, số có dạng $6k \pm 3$ thì chia hết cho 3).

Ta có hàm `isPrime2(n)` dưới đây kiểm tra tính nguyên tố của số n bằng cách kiểm tra xem n có chia hết cho số 2, số 3 và các số có dạng $6k \pm 1$ trong đoạn $[5, \sqrt{n}]$.

```
[]      bool isPrime2(int n)
[]      {
[]           if(n == 2 || n == 3)
[]                return true;
[]           if(n == 1 || n%2 == 0 || n%3 == 0)
[]                return false;
[]           int canN = sqrt(n);
[]           int k = -1;
[]           do
[]           {
[]                k+=6;
[]                if( n%k == 0 || n%(k+2) == 0)
[]                     break;
[]           }while(k<=canN);
[]           return (k > canN);
[]      }
```

2.3 Liệt kê các số nguyên tố trong đoạn [1,n]

Cách 1: Thử lần lượt từng số trong đoạn $[1, n]$ rồi kiểm tra tính nguyên tố của nó.

```
[]      void generate(int n)
[]      {
[]           for( int i=2; i<=n; i++)
[]                if(isPrime2(i))
[]                     cout<<i<<" ";
[]      }
```

Cách 2: Thuật toán sàng nguyên tố Eratosthenes
Dựa theo lý thuyết về số nguyên tố: Một số nguyên tố là số chỉ có 2 ước là 1 và chính nó. Do vậy, nếu ta xác định được số x là số nguyên tố, ta có thể kết luận mọi số chia hết cho x đều không phải số nguyên tố. Do đó ta đã loại bỏ được rất nhiều số mà không cần kiểm tra.
Ví dụ:
Số 2 là số nguyên tố $\Rightarrow$ các số 4, 6, 8, 10, ... không phải số nguyên tố.
Số 3 là số nguyên tố $\Rightarrow$ các số 9, 15, 21, ... không phải số nguyên tố. (Do 12, 18 đã bị loại ở số 2)
Thuật toán sàng nguyên tố Eratosthenes được thực hiện như sau:

1. Tạo mảng đánh dấu cho tất cả các phần tử từ 2 đến N và đặt mặc định tất cả đều là số nguyên tố.

2. Xét số đầu tìm được là số nguyên tố - giả sử x, đánh dấu tất cả các ước của x : $x \times x$, $x \times (x + 1)$, $x \times (x + 2)$, ... trong đoạn $[x, N]$ không phải số nguyên tố.

3. Tìm số tiếp theo được đánh dấu là số nguyên tố trong $[x, N]$. Nếu không còn số nào, thoát chương trình. Nếu còn, gán nó bằng x, và lặp lại bước 2.

4. Khi kết thúc giải thuật, các số không bị đánh dấu là các số nguyên tố.

```
[]      void eratosthenes(int n)
[]      {
[]           bool check[n+1];
[]           //Khởi tạo tất cả các số nguyên tố [2..N] đều là số nguyên tố
[]           for(int i = 2; i<= n; i++)
[]                check[i] = true;
[]           //Thuật toán sàng nguyên tố
[]           //Nếu một số là số nguyên tố
[]           //Thì tất cả các bội của nó không phải là số nguyên tố
[]           for(int i=2; i*i<=n; i++)
[]              if(chec[i] == true)
[]                 for(int j= i*i; j<=n; j+=i)
[]                    check[j]=false;
[]
[]           //In ra các số nguyên tố
[]           for(int i=2; i<n; i++)
[]                if(check[i] == true)
[]                     cout<<i<<" ";
[]      }
```

3 Ước số, bội số

3.1 Số các ước số của một số

Giả sử số N được phân tích thành thừa số nguyên tố như sau:

$$N = a^i \times b^j \times \ldots \times c^k$$

Khi đó 1 ước số N có dạng : $a^x \times b^y \times ... \times c^z$
Trong đó $(0 \leq x \leq i; 0 \leq y \leq j; ...; 0 \leq z \leq k)$
Do đó, số các ước số của N là: $(i+1) \times (j+1) \times ... \times (k+1)$
Ví dụ $120 = 2^3.3.5 \Rightarrow$ Số lượng các ước của 120 là: $(3+1)(1+1)(1+1) = 16$ ước.

3.2 Tổng các ước số của một số

Giả sử số N được phân tích thành thừa số nguyên tố:

$$N = a^i \times b^j \times ... \times c^k$$

Đặt $N1 = b^j \times ... \times c^k$
Gọi $F(N)$ là tổng các ước của N ta có:

$$F(N) = F(N1) + a \times F(N1) + ... + a^i \times F(N1)$$

$$= (1 + a + ... + a^i) \times F(N1) = \frac{(a^i - 1)}{a - 1} \times F(N1)$$

$$= \frac{(a^{i+1} - 1)}{a - 1} \times \frac{(b^{j+1} - 1)}{b - 1} \times ... \times \frac{(c^{k+1} - 1)}{c - 1}$$

Ví dụ $24 = 2^3.3$

Tổng các ước của 24 là:

$$\frac{(2^{3+1}-1)}{2-1} \times \frac{(3^{1+1}-1)}{3-1} = 60$$

3.3 Ước chung lớn nhất của hai số

Ước chung lớn nhất (UCLN) của hai số là số lớn nhất trong tập hợp các ước chung của 2 số đó.
UCLN được tính theo thuật toán Euclid : $UCLN(a,b) = UCLN(b, a \ mod \ b)$

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

3.4 Bội chung nhỏ nhất của hai số

Bội chung nhỏ nhất (BCNN) của hai số là số nhỏ nhất khác 0 trong tập hợp các bội chung của hai số đó.
BCNN được tính theo công thức:

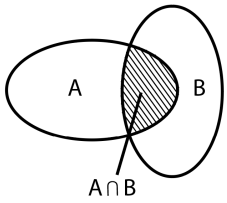
$$BCNN(a,b) = \frac{a \times b}{UCLN(a,b)} = \frac{a}{UCLN(a,b)} \times b$$

4 Lý thuyết tập hợp

Tập hợp là một trong các khái niệm cơ bản của Toán học. Khái niệm tập hợp không được định nghĩa mà chỉ được mô tả qua các ví dụ: Tập hợp các học sinh trong lớp học, tập hợp các cầu thủ trong đội bóng.

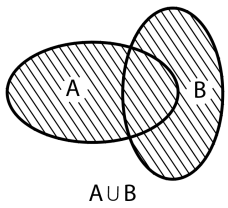
4.1 Các phép toán trên tập hợp

1. Giao của A và B, kí hiệu $A \cap B$, là tập hợp các phần tử đồng thời thuộc cả A và B:



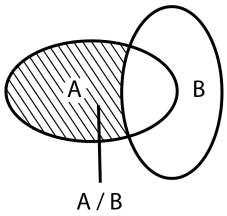
$$A \cap B = \{x : x \in A \text{ và } x \in B\}$$

2. Hợp của A và B, ký hiệu $A \cup B$, là ký hiệu các phần tử hoặc thuộc A hoặc thuộc B:



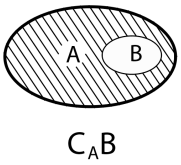
$$A \cup B = \{x : x \in A \text{ hoặc } x \in B\}$$

3. Hiệu của A và B, kí hiệu là $A \setminus B$ là tập hợp các phần tử thuộc tập A nhưng không thuộc B:



$$A \setminus B = \{x : x \in A \text{ và } x \notin B\}$$

4. Phần bù của B trong A, kí hiệu C_AB, là tập hợp các phần tử của A không thuộc B:



$$C_AB = \{x \in A : x \notin B\}$$

4.2 Các tính chất của phép toán trên tập hợp

1. Kết hợp

$$(A \cup B) \cup C = A \cup (B \cup C)$$
$$(A \cap B) \cap C = A \cap (B \cap C)$$

2. Giao hoán

$$A \cap B = B \cap A$$
$$A \cup B = B \cup A$$

3. Phân bố

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$
$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$

4. Đối ngẫu

$$\overline{A \cup B} = \overline{A} \cap \overline{B}$$
$$\overline{A \cap B} = \overline{A} \cup \overline{B}$$

4.3 Tích Đề-các của tập hợp

Tích Đề-các ghép hai tập hợp:

$$A \times B = \{(a, b)|a \in A, b \in B\}$$

Tích Đề-các mở rộng ghép nhiều tập hợp:

$$A_1 \times A_2 \times ... \times A_k = \{(a_1, a_2, ..., a_k)|a_i \in A_i, i = 1, 2, ..., k\}$$

4.4 Nguyên lý cộng

Nếu A và B là hai tập hợp rời nhau thì

$$|A \cup B| = |A| + |B|$$

Nguyên lý cộng mở rộng cho nhiều tập hợp đôi một rời nhau:
Nếu $\{A_1, A_2, ..., A_k\}$ là một phần hoạch của tập X thì:

$$|X| = |A_1| + |A_2| + \cdots + |A_k|$$

4.5 Nguyên lý bù trừ

Nếu A và B không rời nhau thì

$$|A \cup B| = |A| + |B| - |A \cap B|$$

Nguyên lý mở rộng cho nhiều tập hợp:
Giả sử $\{A_1, A_2, ..., A_m\}$ là các tập hợp hữu hạn:

$$|A_1 \cup A_2 \cup \ldots \cup A_m| = N_1 - N_2 + \cdots + (-1)^{m-1} N_m$$

trong đó N_k là tổng phần tử của tất cả các giao của k tập lý từ m tập đã cho.

4.6 Nguyên lý nhân

Nếu mỗi thành phần a_i của bộ có thứ tự k thành phần $(a_1, a_2, \ldots, a_k)$ nó n_i khả năng lựa chọn $(i = 1, 2, \ldots, k)$, thì số bộ sẽ được tạo ra là tích số của các khả năng này $n_1 \times n_2 \times \ldots n_k$
Một hệ quả trực tiếp của nguyên lý nhân:

$$|A_1 \times A_2 \times \ldots \times A_k| = |A_1| \times |A_2| \times \ldots \times |A_k|$$

4.7 Chỉnh hợp lặp

Xét tập hữu hạn gồm n Phần tử $A = \{a_1, a_2, \ldots, a_n\}$.
Một chỉnh hợp lặp chập k của n phần tử là một bộ có thứ tự gồm k phần tử của A các phần tử có thể lặp lại. Một chỉnh hợp lặp chập k của n có thể xem như một phần tử của tích Đề-các A^k. Theo nguyên lí nhân, số tất cả các chỉnh hợp lặp chập k của n sẽ là n^k.

$$\bar{A}_n^k = n^k$$

4.8 Chỉnh hợp không lặp

Một chỉnh hợp không lặp chập k của n phần tử ($k \leq n$) là một bộ có thứ tự gồm k thành phần lấy từ n phần tử của tập hợp đã cho. Các thành phần không được lặp lại. Để xây dựng một chỉnh hợp không lặp, ta xây dựng dần từng thành phần đầu tiên. Thành phần này có n khả năng lựa chọn. Mỗi thành phần tiếp theo, số khả năng lựa chọn giảm đi 1 so với thành phần đứng trước, do đó theo nguyên lý nhân, số chỉnh hợp không lặp chập k của n sẽ là $n(n-1)(n-2)\dots(n-k+1)$.

$$A_n^k = n(n-1)(n-2)\dots(n-k+1) = \frac{n!}{(n-k)!}$$

4.9 Hoán vị

Mỗi hoán vị của n phần tử là một cách xếp thứ tự các phần tử đó. Một hoán vị của n phần tử được xem như một trường hợp riêng của chỉnh hợp không lặp khi $k = n$. Do đó số hoán vị của n phần tử là $n!$.

4.10 Tổ hợp

Một tổ hợp chập k của n phần tử ($k \leq n$) là một bộ không kể thứ tự gồm k thành phần khác nhau lấy từ n phần tử của tập đã cho.

$$C_n^k = \frac{n(n-1)(n-2)\dots(n-k+1)}{k!} = \frac{n!}{k!(n-k)!}$$

Một số tính chất:

- $C_n^0 = C_n^n = 1$
- Tính cách đều : $C_n^k = C_n^{n-k}$
- Hằng đẳng thức pascal $C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$(với $0 < k < n$)

5 Số Fibonacci

5.1 Lý thuyết

Dãy Fibonacci là dãy vô hạn các số tự nhiên bắt đầu bằng hai phần tử 1 và 1, các phần từ sau đó được thiết lập theo quy tắc mỗi phần tử luôn bằng tổng hai phần tử trước đó. Số Fibonacci được xác định bởi công thức sau:

$$F = \begin{cases} F_1 = 1 \\ F_2 = 2 \\ F_n = F_{n-1} + F_{n-2} \quad \text{với } n > 2 \end{cases}$$

Một số phần tử đầu tiên: $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \dots$
Số Fibonacci là đáp án của các bài toán:

a) Bài toán cổ về việc sinh sản của các cặp thỏ như sau:

- Các con thỏ không bao giờ chết.
- Hai tháng sau khi ra đời, mỗi cặp thỏ sẽ sinh ra một cặp thỏ con (một đực, một cái).
- Khi đã sinh con rồi thì cứ mỗi tháng tiếp theo chúng lại sinh được một cặp thỏ con mới.
- Giả sử từ đầu tháng 1 có một cặp mới ra đời thì đến tháng thứ n sẽ có bao nhiêu cặp.

Ví dụ, n = 5, ta thấy:

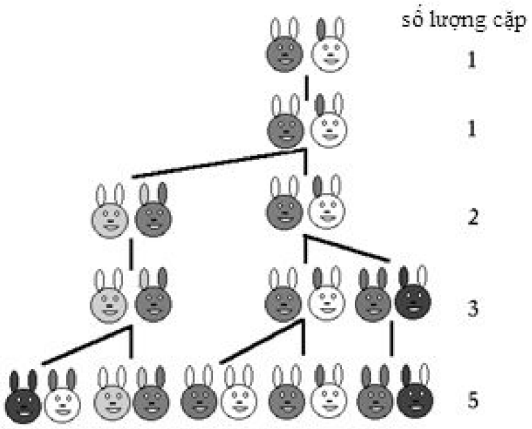
Giữa tháng thứ 1: 1 cặp (cặp ban đầu)

Giữa tháng thứ 2: 1 cặp (ban đầu vẫn chưa đẻ)

Giữa tháng thứ 3: 2 cặp (cặp ban đầu đẻ ra thêm 1 cặp con)

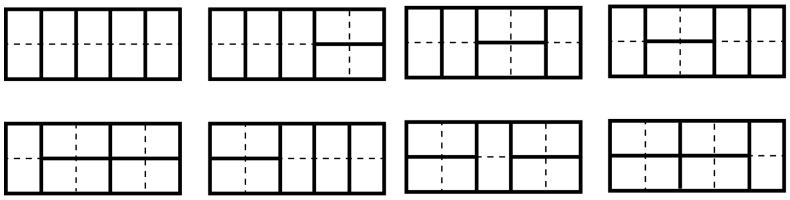
Giữa tháng thứ 4: 3 cặp (cặp ban đầu tiếp tục đẻ)

Giữa tháng thứ 5: 5 cặp



b) Đếm số các xếp $n-1$ thanh DOMINO có kích thước 2×1 phủ kín bảng có kích thước $2 \times (n-1)$.

Ví dụ: Có tất cả 8 cách khác nhau để xếp các thanh DOMINO có kích thước 2× phủ kín bảng 2×5 ($n = 6, Fibonacci_6 = 8$).



5.2 Cách tính số fibonacci

Hàm tính số Fibonacci thứ n bằng phương pháp lặp sử dụng công thức

$$F_n = F_{n-1} + F_{n-2} \text{ với } n \geq 2 \text{ và } F_0 = 0, F_1 = 1$$

```
[]      int Fibo( int n)
[]      {
[]         int fi = 0, fi_1 = 1, fi_2;
[]         for(int i = 2; i<=n; i++)
[]         {
[]               fi_2 = fi + fi_1;
[]               fi = fi_1;
[]               fi_1 = fi_2;
[]          }
[]          return fi_2;
[]      }
[]      int main()
[]      {
[]          for(int i=2; i<20; i++)
[]              cout<<Fibo(i)<<" ";
[]      }
```

6 Số Catalan

Số Catalan là dãy các số tự nhiên xuất hiện trong nhiều bài toán đếm, thường bao gồm những đối tượng đệ quy.
Số Catalan thứ n được xác định:

$$C_n = \frac{1}{n+1}\begin{pmatrix} 2n \\ n \end{pmatrix} = \frac{(2n)!}{(n+1)!n!} = \prod_{k=2}^{n}\frac{n+k}{k} \text{ với } n \geq 0$$

Một số phần tử đầu tiên:$1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, \ldots$
Số Catalan là đáp án của các bài toán

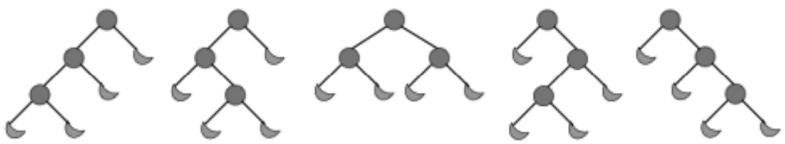
1. Có bao nhiêu cách khác nhau đặt n dấu ngoặc mở và n dấu ngoặc đóng đúng đắn ?

   Ví dụ: $n = 3$ ta có 5 cách sau:

   $$\big(\big(\big(\ \big)\big)\big), \big(\big(\ \big)\big(\ \big)\big), \big(\big(\ \big)\big)\big(\ \big), \big(\ \big)\big(\big(\ \big)\big), \big(\ \big)\big(\ \big)\big(\ \big)$$

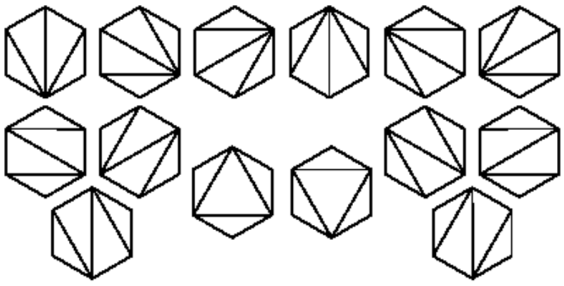
2. Có bao nhiêu cây nhị phân khác nhau có đúng $(n + 1)$ lá ?

   Ví dụ $n = 3$

3. Cho một đa giác lồi $(n + 2)$ đỉnh, ta chia đa giác thành các tam giác bằng cách vẽ các đường chéo không cắt nhau trong đa giác. Hỏi có bao nhiêu cách chia như vậy ?

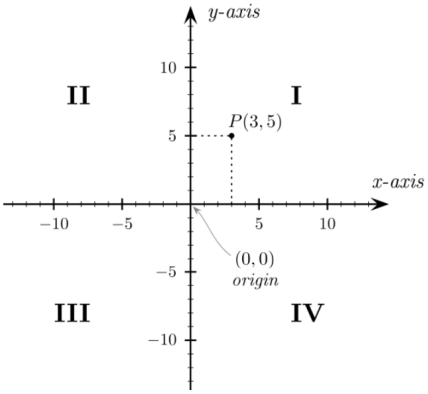
   Ví dụ: $n = 4$



7 Hình học

7.1 Hệ tọa độ Descartes (Đề-các)

Một Hệ tọa độ Descartes xác định vị trí của một điểm trên một mặt phẳng cho trước bằng một cặp số tọa độ (x, y). Trong đó, x và y là 2 giá trị được xác định bởi 2 đường thẳng có hướng vuông góc với nhau (cùng đơn vị đo). 2 đường thẳng đó gọi là trục tọa độ (coordinate axis) (hoặc đơn giản là trục); trục nằm ngang gọi là trục hoành, trục đứng gọi là trục tung; điểm giao nhau của 2 đường gọi là gốc tọa độ (origin) và nó có giá trị là (0, 0).



Hình II.1: coordinate axis

Trên mặt phẳng tọa độ

- Mỗi điểm M xác định một cặp số (x_0, y_0), và ngược lại.
- Cặp số (x_0, y_0) gọi là tọa độ của điểm M, x_0 là hoành độ, y_0 là tung độ của điểm M
- VD: 1 điểm P có tọa độ $(3, 5)$ được kí hiệu là $P(3, 5)$

Trong C++

Nếu chỉ cần lưu một điểm thì có thể lưu bằng 2 biến, nhưng đối với những bài toán cần phải lưu rất nhiều điểm thì chúng ta cần phải định nghĩa 1 điểm trong C++, bằng những cách sau :

```
[]      //Cách 1
[]      pair<int, int> a;
[]      //Cách 2
[]      struct point {
[]         int x, y;
[]      };
[]      point b;
[]      int main(){
[]         cin>>a.first>>a.second; // Nhập bằng cách 1
[]         cin>>b.x>>b.y;     // Nhập bằng cách 2
[]      }
```

7.2 Vectơ

7.2.1 Định nghĩa

Vectơ là một đoạn thẳng có hướng. Vectơ có điểm đầu A, điểm cuối B được kí hiệu $\overrightarrow{AB}$
Nếu điểm $A(x_A; y_A)$ điểm $B(x_B; y_B)$ thì tọa độ của $\overrightarrow{AB} = (x_B - x_A; y_B - y_A)$
Trong C++,với vector $\vec{u} = (x, y)$. Thì ta có thể biểu diễn 2 biến x, y của vectơ giống như cách biểu diễn một điểm ở trên.

7.2.2 Tích vô hướng của hai vectơ

1. **Định nghĩa**

   Cho hai vectơ $\vec{a}$ và $\vec{b}$ khác vectơ $\vec{0}$. Tích vô hướng của $\vec{a}$ và $\vec{b}$ là một số được kí hiệu là $\vec{a}.\vec{b}$, được xác định bởi công thức sau:

   $$\vec{a}.\vec{b} = |\vec{a}|.|\vec{b}| \cos(\vec{a}, \vec{b})$$

2. **Các tính chất của tích vô hướng**

   Người ta chứng minh được các tính chất sau đây của tích vô hướng:

   Với ba vectơ $\vec{a}, \vec{b}, \vec{c}$ bất kì và mọi số k ta có:

   $\vec{a}.\vec{b} = \vec{b}.\vec{a}$ (tính chất giao hoán)

   $\vec{a}.(\vec{b} + \vec{c}) = \vec{a}.\vec{b} + \vec{a}.\vec{c}$ (tính chất phân phối)

   $(k.\vec{a}).\vec{b} = k(\vec{a}, \vec{b}) = \vec{a}.(k\vec{b})$

3. **Biểu thức tọa độ của tích vô hướng**

   Trên mặt phẳng tọa độ $(0; \vec{i}; \vec{j})$, cho hai vectơ $\vec{a} = (a_1; a_2)$, $\vec{b} = (b_1; b_2)$. Khi đó tích vô hướng $\vec{a}$ và $\vec{b}$ là:

   $$\vec{a}.\vec{b} = a_1b_1 + a_2b_2$$

   Nhận xét: Hai vectơ $\vec{a} = (a_1; a_2)$, $\vec{b} = (b_1; b_2)$ khác vectơ $\vec{0}$ vuông góc với nhau khi và chỉ khi:

   $$a_1b_1 + a_2b_2 = 0$$

7.2.3 Tích có hướng của hai vectơ

1. **Khái niệm**

   Tích có hướng là một vectơ là một phép toán nhị nguyên phân trong không gian ba chiều của vectơ. Kết quả thu được là một vectơ có hướng vuông góc với mặt phẳng chứa hai vectơ đầu vào của phép nhân.

2. **Định nghĩa**

   Vectơ $\vec{w}$ được gọi là tích có hướng của hai vectơ $\vec{u} = (x_1; y_1; z_1)$ và $\vec{v} = (x_2; y_2; z_2)$ trong không gian, ký hiệu là $\vec{w} = \vec{u} \wedge \vec{v}$ hoặc $\vec{w} = [\vec{u}, \vec{v}]$ khi đó:

   $\vec{w}$ có phương vuông góc với cả $\vec{u}$ và $\vec{v}$.

   $\vec{w} = (y_1z_2 - y_2z_1; z_1x_2 - z_2x_1; x_1y_2 - x_2y_1)$

3. **Tính chất**

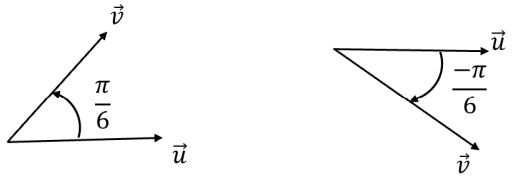
- $\left[\vec{u_1};\vec{u_2}\right] = -\left[\vec{u_2};\vec{u_1}\right]$
- $\left[\vec{u_1};\vec{u_2}\right] = \vec{0} \Leftrightarrow \vec{u_1}$ cùng phương $\vec{u_2}$
- $\left[\vec{u_1};\vec{u_2}\right] \perp \left[\vec{u_1}\right]$; $\left[\vec{u_1};\vec{u_2}\right] \perp \left[\vec{u_2}\right]$
- $\left[\vec{u_1};\vec{u_2}\right].\vec{u_3} = 0 \Leftrightarrow$ ba vectơ $\vec{u_1}, \vec{u_2}, \vec{u_3}$ đồng phẳng.
- $\left|\left[\vec{u_1};\vec{u_2}\right]\right| = \left|\vec{u_1}\right|.\left|\vec{u_2}\right| \sin\left(\vec{u_1}, \vec{u_2}\right)$

4. **Ứng dụng tích có hướng**

- Diện tích tam giác:
$$S_{ABC} = \frac{1}{2}\left|\left[\overrightarrow{AB}, \overrightarrow{AC}\right]\right|$$
- Diện tích hình bình hành:
$$S_{ABCD} = \left|\left[\overrightarrow{AB}, \overrightarrow{AD}\right]\right| = \left|\left[\overrightarrow{AB}, \overrightarrow{AC}\right]\right|$$
- Thể tích tứ diện:
$$V_{ABCD} = \frac{1}{6}\left|\left[\overrightarrow{AB}, \overrightarrow{AC}\right].\overrightarrow{AD}\right|$$
- Thể tích khối hộp:
$$V_{ABCD.A'B'C'D'} = \left|\left[\overrightarrow{AB}, \overrightarrow{AD}\right].\overrightarrow{AA'}\right|$$

7.2.4 Tích chéo của hai vectơ

Tích chéo (cross product) của hai vectơ $\vec{u}$ và $\vec{v}$, kí hiệu $\vec{u} \times \vec{v}$ là một số thực được tính bằng độ dài hai vectơ $\vec{u}$ và $\vec{v}$ nhân với sin góc xen giữa hai vectơ đó. Góc xen giữa hai vectơ này là góc định hướng, có số đo từ $-\pi$ tới π, số đo mang dấu dương nếu chiều quay từ $\vec{u}$ tới $\vec{v}$ là chiều thuận (ngược chiều kim đồng hồ) và mang dấu âm đến chiều quay từ $\vec{u}$ tới $\vec{v}$ là chiều nghịch (theo chiều kim đồng hồ).



Hình II.2: Góc có hướng

Tích chéo là một khái niệm suy ra từ khái niệm tích có hướng trong không gian vectơ Ơclit nhiều chiều. Bằng các công cụ đại số tuyến tính, người ta đã chứng minh được công thức của tích chéo giữa hai vectơ $\vec{u} = (x_1; y_1)$ và $\vec{v} = (x_2; y_2)$:

$$\vec{u} \times \vec{v} = x_1 y_2 - x_2 y_1 = \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix}$$

tức là giá trị của tích chéo bằng định thức của ma trận $\begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix}$.

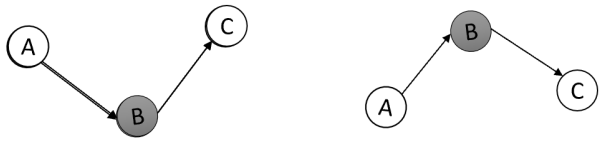
Ta cũng suy ra công thức tính sin của góc định hướng α giữa hai vectơ $\vec{u} = (x_1; y_1)$ và $\vec{v} = (x_2; y_2)$:

$$\sin\alpha = \frac{\vec{u} \times \vec{v}}{|\vec{u}|.|\vec{v}|} = \frac{x_1y_2 - x_2y_1}{\sqrt{(x_1^2 + y_1^2)(x_2^2 + y_2^2)}}$$

Về mặt hình học, giá trị tuyệt đối của tích chéo $\vec{u} \times \vec{v}$ là diện tích hình bình hành $OABC$, trong đó O là gốc tọa độ, $\overrightarrow{OA} = \vec{u}, \overrightarrow{OC} = \vec{v}$ và $\overrightarrow{OB} = \vec{u} + \vec{v}$.

Tích chéo có một số ứng dụng quan trọng trong việc khảo sát chiều: Giả sử ta đi từ điểm A sang điểm B theo đường thẳng và đi tiếp sang điểm C theo đường thẳng, khi đó:

- Tích chéo $\overrightarrow{AB} \times \overrightarrow{BC}$ sẽ là số dương nếu chỗ rẽ tại B là "rẽ trái" (hay gọi đúng hơn là bẻ góc ngược chiều kim đồng hồ);
- Tích chéo $\overrightarrow{AB} \times \overrightarrow{BC}$ là số âm nếu chỗ rẽ tại B là "rẽ phải";
- Tích chéo $\overrightarrow{AB} \times \overrightarrow{BC} = 0$ có nghĩa là 3 điểm A, B, C thẳng hàng.



Hình II.3: Rẽ trái và rẽ phải

Một ví dụ về ứng dụng của tích vô hướng và tích chéo: Trên mặt phẳng cho ba điểm $A = (x_A; y_A), B = (x_B; y_B)$ và $C = (x_C; y_C)$, hãy cho biết điểm C có nằm trên đoạn AB hay không.

Điều kiện cần và đủ để C nằm trên đoạn thẳng AB là A, B, C thẳng hàng và hai vectơ $\overrightarrow{AC}, \overrightarrow{BC}$ không cùng hướng hoặc một số chúng là vectơ $\vec{0}$. Điều kiện này có thể viết bởi

$$\overrightarrow{AC} \times \overrightarrow{BC} = 0 \text{ và } \overrightarrow{AC}.\overrightarrow{BC} \leq 0$$

7.3 Một số bài toán cơ bản hình học

7.3.1 Tính độ dài đoạn thẳng

Trong mặt phẳng tọa độ Oxy, cho hai điểm $A(x_A, y_A)$ và $B(x_B; y_B)$. Độ dài đoạn thẳng AB là:

$$AB = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$$

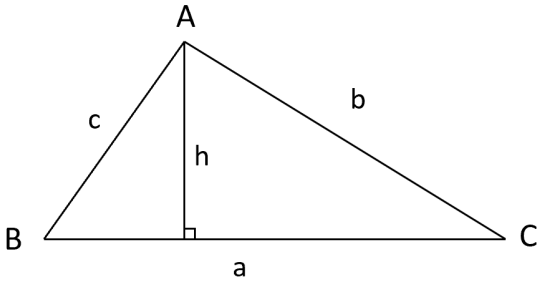
7.3.2 Tính góc giữa 2 vectơ

Trong mặt phẳng với hệ trục tọa độ (Đề-các) vuông góc Oxy. Có hai vectơ $\vec{n_1} = (a_1; b_1)$ và $\vec{n_2} = (a_2; b_2)$ thì cos của góc giữa hai vectơ này được tính theo công thức:

$$\cos\left(\vec{n_1}, \vec{n_2}\right) = \frac{\vec{n_1}.\vec{n_2}}{|\vec{n_1}|.|\vec{n_2}|} = \frac{a_1a_2 + b_1b_2}{\sqrt{a_1^2 + b_1^2}.\sqrt{a_2^2 + b_2^2}}$$

Từ đây, ta có thể suy ra góc giữa của 2 vectơ là : arccos(giá trị của công thức trên).

7.3.3 Tính diện tích tam giác



Hình II.4: Diện tích tam giác

Tùy vào mỗi bài toán, mỗi thông tin được cho mà ta sử dụng các công thức thích hợp để tính diện tích tam giác:

- Khi biết độ dài một cạnh của tam giác a, và chiều cao tương ứng h:

$$S = \frac{a+h}{2}$$

- Khi biết độ dài hai cạnh tam giác là b và c, góc xe giữa hai cạnh đó là α:

$$S = \frac{bc\sin\alpha}{2}$$

- Nếu biết độ dài 3 cạnh là a, b, c. Theo công thức Hê-rông:

$$S = \sqrt{p(p-a)(p-b)(p-c)}$$

  Trong đó: p là nửa chu vi của tam giác: $p = \frac{a+b+c}{2}$

- Biết hai vectơ $\overrightarrow{AB}$ và $\overrightarrow{AC}$ thì tính bằng công thức tích chéo:

$$S = \left|\frac{\overrightarrow{AB} \times \overrightarrow{AC}}{2}\right|$$

  Công thức này nếu biết tọa độ ba điểm A, B, C thì:

$$S = \left|\frac{(x_B - x_A)(y_C - y_A) - (x_C - x_A)(y_B - y_A)}{2}\right|$$

  Chú ý: nếu bỏ giá trị tuyệt đối công thức tính sẽ cho giá trị dương nếu hướng quay từ vectơ $\overrightarrow{AB}$ tới $\overrightarrow{AC}$ là hướng thuận. Ngược lại, công thức sẽ cho giá trị âm nếu hướng quay $\overrightarrow{AB}$ tới $\overrightarrow{AC}$ là hướng nghịch.

7.3.4 Tính diện tích đa giác

Để tính được diện tích đa giác đầu tiên ta phải hiểu về đa giác.
Đa giác là một đường gấp khúc khép kín, không tự cắt. Trong lập trình, một đa giác được lưu bởi một dãy các đỉnh liên tiếp nhau $A_1, A_2, ..., A_n$.
Khi đó diện tích đại số của một đa giác không tự cắt có thể xác định bởi công thức:

$$S = \frac{(x_1 - x_2)(y_1 + y_2) + (x_2 - x_3)(y_2 + y_3) + ... + (x_n - x_1) + (y_n + y_1)}{2}$$

$|S|$ chính là diện tích của đa giác.
Để dễ hình dung có thể theo công thức ngắn gọn hơn, lúc đó quy ước điểm A_{n+1} trùng với điểm A_1.

$$S = \frac{1}{2}\left|\sum_{i=1}^{n}(x_i y_{i+1} - x_{i+1} y_i)\right|$$

$$= \frac{1}{2}\left|\sum_{i=1}^{n}(x_{i-1} - x_{i+1}) y_i\right|$$

Chú ý: Các đỉnh của đa giác phải được lưu cùng hoặc ngược chiều kim đồng hồ.
Ta có thể biết thứ tự lưu đỉnh nhờ đấu của $S : S > 0$ có nghĩa là đỉnh của đa giác được liệt kê ngược chiều kim đồng hồ (và ngược lại).

CHUYÊN ĐỀ III

Sắp xếp

Sắp xếp là quá trình bố trí lại vị trí các đối tượng của một danh sách theo một trật tự nhất định. Sắp xếp đóng vai trò quan trọng trong cuộc sống nói chung và trong tin học nói riêng, thử hình dung xem, một cuốn từ điển, nếu các từ không được sắp xếp theo thứ tự, sẽ khó khăn như thế nào trong việc tra cứu từ ngữ.

Do đặc điểm dữ liệu (kiểu số hay phi số, kích thước bé hay lớn, lưu trữ ở bộ nhớ trong hay bộ nhớ ngoài, truy cập tuần tự hay ngẫu nhiên...) mà người ta có các thuật toán sắp xếp khác nhau. Trong chuyên đề này, chúng ta chỉ quan tâm đến các thuật toán sắp xếp trong trường hợp dữ liệu được lưu trữ ở bộ nhớ trong(nghĩa là toàn bộ dữ liệu cần sắp xếp phải được đưa vào bộ nhớ chính của máy tính).

1 Phát biểu bài toán

Giả sử các đối tượng cần sắp xếp được biểu diễn bởi bản ghi gồm một số trường. Một trong các trường đó gọi là *khóa sắp xếp (key)*. Kiểu của khóa là kiểu có thứ tự chẳng hạn (kiểu số nguyên, kiểu số thực, ...)

Bài toán sắp xếp được phát biểu như sau: Cho mảng a các đối tượng, cần sắp xếp lại các thành phần (phần tử) của mảng a để nhận được mảng a mới với các thành phần có các giá trị khóa tăng dần:

$$a[1].key \leq a[2].key \leq \cdots \leq a[n].key$$

2 Các thuật toán sắp xếp thông dụng

2.1 Thuật toán sắp xếp nổi bọt (Bubble Sort)

Ý tưởng cơ bản của thuật toán là lấy ý tưởng từ các bong bóng nước, nếu sắp xếp tăng dần, các phần tử bé sẽ từ từ nổi lên ở đầu mảng.

Cụ thể: Thuật toán thực hiện bằng cách lặp lại công việc đổi chỗ hai số liên tiếp nhau nếu đứng sai vị trí cho đến khi dãy được sắp xếp.

Ví dụ minh họa sắp dãy $[5, 1, 4, 2, 8]$ tăng dần:

Lần đầu tiên:

x[0]	x[1]	x[2]	x[3]	x[4]	so sánh	đổi chỗ
5	**1**	4	2	8	$5 > 1$	có
1	**5**	**4**	2	8	$5 > 4$	có
1	4	**5**	**2**	8	$5 > 2$	có
1	4	2	**5**	**8**	$5 < 8$	không

Lần lặp thứ 2:

x[0]	x[1]	x[2]	x[3]	x[4]	so sánh	đổi chỗ
1	**4**	2	5	8	$1 < 4$	không
1	**4**	**2**	5	8	$4 > 2$	có
1	2	**4**	**5**	8	$4 < 5$	không
1	2	4	**5**	**8**	5 <8	không

Lần lặp thứ 3:

x[0]	x[1]	x[2]	x[3]	x[4]	so sánh	đổi chỗ
1	**2**	4	5	8	5 <8	không
1	**2**	**4**	5	8	5 <8	không
1	2	**4**	**5**	8	5 <8	không
1	2	4	**5**	**8**	5 <8	không

Code sắp xếp nổi bọt:

```
[]   void swap(int &x, int &y)
[]   {
[]      int temp = x;
[]      x = y;
[]      y = temp;
[]   }
[]   // hàm sắp xếp bubble sort
[]   void bubbleSort(int arr[], int n)
[]   {
[]      int i, j;
[]      bool haveSwap = false;
[]      for( i = 0; j < n - 1; i++) {
[]         haveSwap = false;
[]         for(j = 0; j< n - i - 1; j++) {
[]            if(arr[i] > arr[i+1] {
[]               swap(arr[i], arr[i+1]);
[]               //Kiểm tra lần này có swap không
[]               haveSwap = true;
[]            }
[]         }
[]         // Nếu không có swap nào thực hiện
[]         // Thì mảng đã sắp xếp không cần lặp nữa
[]         if( haveSwap == false ) break;
[]      }
[]   }
```

Độ phức tạp của thuật toán:

+ Trường hợp tốt: $O(n)$
+ Trung bình: $O(n^2)$
+ Trường hợp xấu: $O(n^2)$

2.2 Thuật toán sắp xếp QuickSort (Quick sort)

Ý tưởng của thuật toán sắp xếp QuickSort là thuật toán dựa trên kĩ thuật chia để trị. Thuật toán được mô tả như sau:

- **Trường hợp cơ sở:** nếu dãy chỉ còn không quá một phần tử thì dãy nó đã được sắp xếp nên không cần thêm gì cả.
- **Chia:** ở thao tác này gồm 2 công việc:
  - Chọn 1 phần tử trong dãy làm phần tử chốt p (pivot).
  - Phân đoạn: chia dãy đã cho thành 2 dãy con, dãy con trái(L) sẽ gồm những phần tử nhỏ hơn phần tử chốt và dãy con phải(R) là những phần tử lớn hơn phần tử chốt.
- **Trị:** lặp lại một cách đệ quy thuật toán với 2 dãy con trái và phải.
- **Tổng hợp:** dãy được sắp xếp L, q, R.

Như vậy đối với thuật toán QuickSort thì việc chọn phần từ chốt có vai trò quyết định đối với hiệu quả của thuật toán. Và người ta dùng các cách sau đây làm phần tử chốt:

– Chọn phần tử đầu tiên hoặc đứng cuối làm phần tử chốt

– Chọn phần tử đứng giữa làm phần tử chốt

– Chọn phần tử trung vị 3 phần tử đứng đầu, đứng cuối và đứng giữa làm phần tử chốt

– Chọn phần tử ngẫu nhiên làm phần tử chốt

Các bước của thuật toán với trường hợp chọn chốt là phần tử cuối :

Bước 1: Lấy phần tử chốt là phần tử ở cuối danh sách.

Bước 2: Chia mảng theo phần tử chốt.

Bước 3: Sử dụng sắp xếp nhanh một cách đệ quy với mảng con bên trái.

Bước 4: Sử dụng sắp xếp nhanh một cách đệ quy với mảng con bên phải.

Code thuật toán Quick sort:

```
[] int partition (int arr[], int low, int high) {
[]    int pivot = arr[high];
[]    int left = low;
[]    int right = high-1;
[]    while( true ) {
[]       //Tìm phần tử >= arr[pivot]
[]       while( left <= right && arr[left] < piviot ) left++;
[]       //Tìm phần tử <= arr[pivot]
[]       while(right >= left && arr[right] > pivot) right--;
[]       if (left >= right) break; //Đã duyệt xong thì thoát khỏi vòng lặp
[]       swap(arr[left], arr[right]) //Nếu chưa xong thì đổi chỗ
[]       left++; //Vì left hiện tại đã xét, nên cần tăng
```

```
[]      right--; //Vì right hiện tại đã xét, nên cần giảm
[]   }
[]   swap(&arr[left], arr[high]);
[]   return left; // Trả về chỉ số sẽ dùng để chia đôi mảng;
[] }
[] void quickSort(int arr[], int low, int high) {
[]   if( low < high ) {
[]      //pi là chỉ số nơi phần tử này đã đứng đúng vị trí
[]      //và là phần tử chia mảng làm 2 mảng con trái và phải
[]      int pi = partition(arr, low, high);
[]      // Gọi đệ  quy sắp xếp 2 mảng con trái và phải
[]      quickSort(arr, low, pi-1);
[]      quickSort(arr, pi+1, high);
[]   }
[] }
```

Code đơn giản hơn khi chọn điểm chốt là phần tử ở giữa đoạn đang xét:

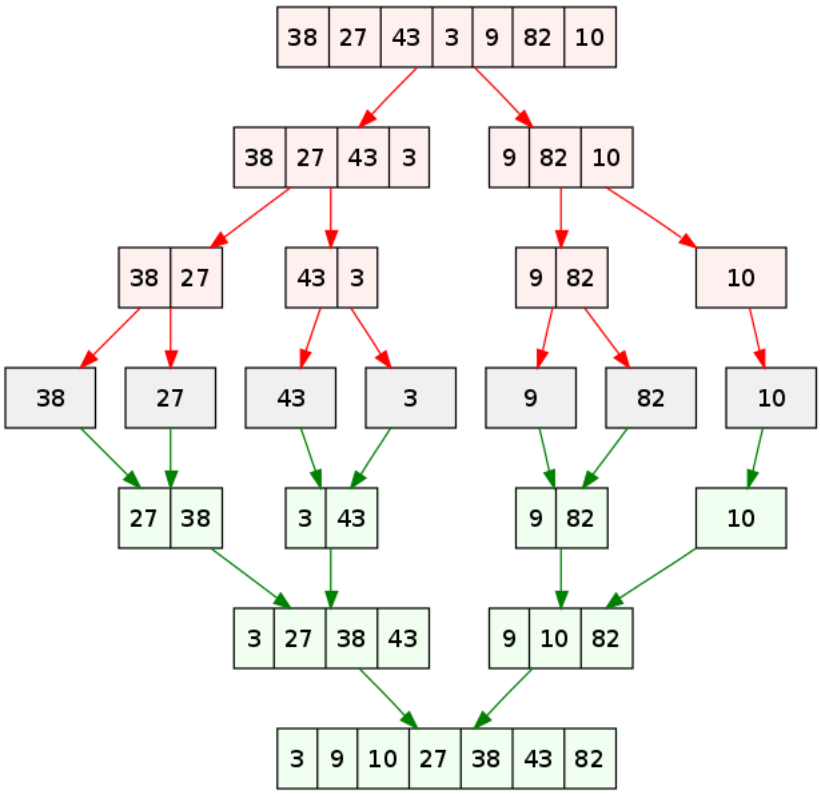
```
[]   void qsort(int a[], int l, int r) {
[]      if(l >= r ) return;
[]      int i = l,
[]         j = r,
[]         x = a[(l+r)/2];
[]      while( i <= j) {
[]         while (a[i] < x) i++;
[]         while (a[j] > x) j--;
[]         if(i <= j ) swap(a[i++], a[j--]);
[]      }
[]      qsort(a, l, j);
[]      qsort(a, i, r);
[]   }
```

Đánh giá độ phức tạp:
Độ phức tạp của thuật toán phụ thuộc nhiều vào phần tử chốt nên độ phức tạp cũng phụ thuộc vào đó mà khác nhau:

+ Trường hợp tốt: $O(nlog(n))$
+ Trung bình: $O(nlog(n))$
+ Trường hợp xấu: $O(n^2)$

2.3 Thuật toán sắp xếp trộn (Merge sort)

Giống như Quick sort, Merge Sort là một thuật toán chia để trị. Thuật toán này chia mảng cần sắp xếp thành 2 nửa. Tiếp tục lặp lại việc này ở các nửa mảng đã chia. Sau cùng gộp các nửa đó thành mảng đã sắp xếp. Hàm merge() được sử dụng để gộp hai nửa mảng. Hàm merge(arr, l, m, r) là tiền trình quan trọng nhất sẽ gộp hai nửa mảng thành một mảng sắp xếp, các nửa mảng là arr[l...m] và arr[m+1...r] sau khi gộp sẽ thành một mảng duy nhất đã sắp xếp.
Hàm gồm thuật toán hoạt động như sau: Giả sử ta có hai mảng đã sắp xếp a[1...m] và b[1...n]. Ta gộp chúng thành một mảng mới c[1...m+n] theo cách sau:



Hình III.1: ảnh minh họa sắp xếp một mảng theo merge sort

- So sánh hai phần tử đứng đầu của hai mảng, lấy phần tử nhỏ hơn cho vào mảng mới. Tiếp tục như vậy cho tới khi một trong hai mảng rỗng.
- Khi một trong hai mảng là rỗng ta lấy phần còn lại của mảng kia cho vào cuối mảng mới.

Ví dụ: Cho hai mảng $a = 1, 3, 7, 9$ $b = 2, 4$, quá trình gộp diễn ra như sau:

Mảng A	Mảng B	So Sánh	Mảng C
1, 3, 7, 9	2, 6	$1 < 2$	1
3, 7, 9	2, 6	3>2	1, 2
3, 7, 9	6	3<6	1, 2, 3
7, 9	6	7>6	1, 2, 3, 6
7, 9	NULL		1, 2, 3, 6, 7, 9

Code của thuật toán merge sort:

```
[] //Hợp nhất hai mảng đầu tiên của A[]
[] //Mảng đầu tiên là A[l..m]
[] //Mảng thứ hai là A[m+1..r]
[] void merge(int A[], int l, int m, int r) {
[]    int n1 = m - l - 1; // Độ dài mảng thứ nhất (mảng L[])
[]    int n2 = r - m; // Độ dài mảng thứ hai (mảng R[])
[]    int L[n1], R[n2];
[]    // sao chép từ mảng A sang hai mảng L[] và R[]
[]    for(int i = 0; i < n1; i++)   L[i] = A[i + l];
[]    for(int i = 0; i < n2; i++) R[i] = A[i + m + 1];
[]       int i = 0, // i là chỉ số của mảng L
```

```
[]      j = 0, // j là chỉ số của mảng R
[]      k = l, // k là chỉ số của mảng ban đầu
[]   //hợp nhất hai mảng L[] va R[] thành A[l..r]
[]   while(i < n1 && j < n2) {
[]      if(L[i] <= R[i]) {
[]         A[k] = L[i];
[]         i++;
[]      }else {
[]         A[k] = R[j];
[]         j++;
[]      }
[]   }
[]   while(i < n1) {// sao chép các phần tử còn lại của mảng L
[]      A[k] = L[i];
[]      i++;
[]      k++;
[]   }
[]   while(j < n2) {// sao chép các phần tử còn lại của mảng R
[]      A[k] = R[i];
[]      j++;
[]      k++;
[]   }
[] }
[] //hàm sắp xếp chính dùng đệ quy
[] void mergeSort(int A[], int l, int r) {
[]   if(l < r) {
[]      int m = ( l + r) / 2;
[]      mergeSort(A, l, m);
[]      mergeSort(A, m+1, r);
[]      merge(A, l, m, r);
[]   }
[] }
[] int main() {
[]   int A[] = { 12, 11, 13, 5, 6, 7};
[]   int n = sizeof(A) / sizeof(A[0]);
[]   mergeSort(A, 0, n-1);
[] }
```

Đánh giá độ phức tạp:
Merge Sort có độ phức tạp là $O(nlog(n))$ trong cả ba trường hợp (xấu nhất, trung bình và tốt nhất) vì sắp xếp này luôn chia mảng thành hai nửa và mất thời gian tuyến tính để hợp nhất hai nửa.

3 Sắp xếp bằng đếm phân phối (Counting sort)

Giả sử mảng có N phần tử cần sắp xếp các giá trị của mảng đều là số nguyên và thuộc khoảng [0...M].

Ý tưởng của thuật toán là đếm các số trong khoảng [0...M].

- Có bao nhiêu giá trị 0(Giả sử có $C(0)$ giá trị).

- Có bao nhiêu giá trị 1(Giả sử có $C(1)$ giá trị).
- ...
- Có bao nhiêu giá trị M(Giả sử có $C(M)$ giá trị).

Sau đó sắp xếp lại mảng bằng cách đặt $C(0)$ phần tử 0 ở đầu, đặt $C(1)$ phần tử 1 tiếp theo, ... và đặt $C(M)$ phần tử M cuối cùng.
Ví dụ: Cho dãy $a = 2, 1, 3, 1, 2$
- Giá trị 2 đứng ở vị trí 1 : $C(2) = 1$.
- Giá trị 1 đứng ở vị trí 2 : $C(1) = 1$.
- Giá trị 3 đứng ở vị trí 3 : $C(3) = 1$.
- Giá trị 1 đứng ở vị trí 4 : $C(2) = 2$.
- Giá trị 2 đứng ở vị trí 5 : $C(2) = 2$.

Nên ta được $C(1) = 2$, $C(2) = 2$, $C(3) = 1$. Vậy mảng sau khi sắp xếp: 1 1 2 2 3.
Code thuật toán Counting sort:

```
[]   void CountingSort(int arr[], int n) {
[]      int mi=arr[0],mx=arr[0];
[]      // tìm min, max của mảng
[]      for(int i = 1; i < n; i++) {
[]         mi = min(arr[i], mi);
[]         ma = max(arr[i], ma);
[]      }
[]      int d = 0; cs = mx - mi;
[]      // mảng lưu kết quả đếm
[]      int count = new int[cs+1];
[]      // khởi tạo giá trị của mỗi phần tử trong mảng đếm là 0
[]      for( int i = 0; i<=cs; i++)
[]         count[i] = 0;
[]      // tìm số lần xuất hiện của giá trị trong mảng chỉnh
[]      for(int i = 0; i < n; i++)
[]         count[arr[i]-mi]++;
[]
[]      // gán giá trị mảng đếm vào mảng chính
[]      // Vì mảng đã gán xong nên có thể viết đè được
[]      for(int i = 0; i <= cs; i++)
[]         if(count[i] > 0)
[]            for(int j = 1; j<= count[i]; j++)
[]               arr[d++] = i + mi;
[]   }
```

Đánh giá độ phức tạp:
Độ phức tạp của thuật toán này là $O(max(N, M))$. Giá trị của M càng nhỏ thì thuật toán sắp xếp càng nhanh.

CHUYÊN ĐỀ IV

Thiết kế giải thuật

Chuyên đề này trình bày các chiến lược thiết kế như: Quay lui(Backtracking), Nhánh và cận(Branch and Bound), Tham lam(Greedy Method), Chia để trị(Divide and Conquer) và Quy hoạch động(Dynamic Programming). Đây là các chiến lược tổng quát, nhưng mỗi phương pháp chỉ áp dụng cho mỗi chiến lược nhất định chỉ áp dụng được cho một số bài toán nhật định, chức không có một phương pháp nào vạn năng để giải quyết mọi bài toán. Các phương pháp thiết kế thuật toán trên là chiến lược, có tính định hướng tìm thuật toán. Việc áp dụng chiến lược để tìm ra thuật toán đòi hỏi nhiều sáng tạo, nên trong chuyên đề này, ngoài phân tích trình bày các phương pháp, chuyên đề còn có những ví dụ cụ thể để, cùng với giải thuật và cài đặt, để có cái nhìn chi tiết từ việc thiết kế giải thuật đến xây dựng chương trình.

1 Quay lui (Backtracking)

Quay lui, vét cạn, thử sai, duyệt ... là một số tên gọi tuy không đồng nghĩa nhưng cùng chỉ một phương pháp trong tin học: tìm nghiệm của một bài toán bằng cách xem xét tất cả các phương án có thể. đối với con người phương pháp này thường là không khả thi vì số phương án cần kiểm tra lớn. Tuy nhiên đối với máy tính, nhờ tốc độ xử lý nhanh, máy tính có thể giải rất nhiều bài toán bằng phương pháp quay, lui vét cạn.
Ưu điểm của phương pháp quay lui, vét cạn là luôn đảm bảo *tìm ra nghiệm đúng, chính xác*. Tuy nhiên, hạn chế của phương pháp này là thời gian thực thi lâu, độ phức tạp lớn. Do đó vét cạn thường chỉ phù hợp với các bài toán có kích thước nhỏ.

1.1 Phương pháp

Dùng để giải bài toán liệt kê các cấu hình. Mỗi cấu hình được xây dựng bằng từng phần tử. Mỗi phần tử lại được chọn bằng cách thử tất cả các khả năng.
Các bước trong việc liệt kê cấu hình dạng $X[1...n]$:
- Xét tất cả các giá trị $X[1]$ có thể nhận, thử $X[1]$ nhận các giá trị đó. Với mỗi giá trị của X[1] ta sẽ:
- Xét tất cả giá trị $X[2]$ có thể nhận, lại thử $X[2]$ cho các giá trị đó. Với mỗi giá trị X[2] lại xét khả năng giá trị của $X[3]$...tiếp tục như vậy cho tới bước:
- ...
- Xét tất cả giá trị X[n] có thể nhận, thử cho $X[n]$ nhận lần lượt giá trị đó.
- Thông báo cấu hình tìm được.

Bản chất của quay lui là một quá trình tìm kiếm theo chiều sâu(Depth-First Search).
Mô hình thuật toán:

```
[]   Backtracking(k) {
[]      for([Mỗi phương án chọn i(thuộc tập D)]) {
[]         if([Chấp nhận i]) {
[]            <Chọn i cho X[k]>;
[]            if([Thành công]) {
[]               <Đưa ra kết quả>;
[]            } else {
[]               Backtracking(k+1);
[]               <Bỏ chọn i>;
[]            }
[]         }
[]      }
[]   }
```

1.2 Bài toán N quân hậu.

Cần đặt N quân hậu vào bàn cờ vua $N \times N$, sao cho chúng không tấn công nhau, tức là không có hai quân hậu nào cùng hàng, cùng cột hoặc cùng đường chéo.

Ý tưởng: là đặt từng quân hậu trong các cột khác nhau, bắt đầu từ cột ngoài cùng bên trái. Khi đặt một quân hậu vào một cột, ta kiểm tra các cuộc đụng độ với các quân hậu đã được đặt. Trong cột hiện tại, nếu chúng ta tìm thấy một hàng không có xung đột, chúng ta đánh dấu hàng và cột này là một phần của giải pháp. Nếu chúng ta không tìm thấy một hàng như vậy do đụng độ thì chúng ta quay lại và *return false*.

```
[]   bool isSafe(int board[N][N], int row, int col) {
[]      int i,j;
[]      for(i = 0; i < col; i++)
[]         if(board[row][i])
[]            return false;
[]      for(i = row, j = col; j >= 0 && j >= 0; i--, j--)
[]         if(board[i][j])
[]            return false;
[]      for(i = row, j = col; j >= 0 && i < N; i++, j--)
[]         if(board[i][j])
[]            return false;
[]      return true;
[]   }
[]   bool solveNQutil(int board[N][N], int col) {
[]      if (col >= N)
[]         return true;
[]      for(int i = 0; i <= N; i++)
[]         if(isSafe(board, i, col)) {
[]            board[i][col]=1;
[]            if(solveNQutil(board,col+1))
[]               return true;
[]            board[i][col] = 0; // BACKTRACK
[]         }
[]      }
```

```
[]      return false;
[]   }
```

Ta có thể tối ưu hàm **isSafe() :**

Ý tưởng là không kiểm tra mọi phần tử theo đường chéo phải và trái thay vào đó hãy sử dụng thuộc tính của đường chéo:

1. Tổng của i và j là không đổi và duy nhất cho mỗi đường chéo bên phải, trong đó i là hàng của phần tử và j là cột của phần tử.

2. Sự khác biệt của i và j là không đổi và duy nhất đối với mỗi đường chéo bên trái, trong đó i và j lần lượt là hàng và cột của phần tử.

```
[]   // ld là một mảng lưu các chỉ số hiển thị row - col + N + 1
[]   // (N - 1) là để chuyển chênh lệch không bị âm
[]   int ld[30] = {0};
[]   // rl là một mảng trong đó có các chỉ số của nó biểu thị row + col
[]   // và được sử dụng để kiểm tra xem 1 Queen có thể được đặt trên
[]   // đường chéo phải hay không
[]   int rd[30] = {0};
[]   // cl là một mảng trong đó các chỉ số của nó cho biết col và
[]   // được sử dụng để kiểm tra xem 1 Queen có thể được đặt
[]   // trong col đó hay không
[]   int cl[30] = {0};
[]
[]   bool solveNQUtil(int board[N][N], int col) {
[]      if(col >= N)
[]         return true;
[]      for(int i = 0; i < N; i++) {
[]         // Kiểm tra xem Queen có thể đặt trên board[i][col]
[]         // Kiểm tra xem một Queen có thể được đặt trên board[row][col]
   không.
[]         // Chúng ta chỉ cần kiểm tra ld[row - col + n -1] và rd[row + col]
[]         // trong đó ld và rd là trái và phải đường chéo tương ứng
[]         if((ld[i - col + N - 1] != 1 &&
[]            rd[i + col] != 1) && cl[i] != 1) {
[]         // Đặt Queen trên board[i][col]
[]            board[i][col] = 1;
[]            ld[i - col + N - 1] = rd[i + col] = cl[i] = 1
[]            if(solveNQUtil(board, col + 1))
[]               return true;
[]            board[i][col] = 0; // BACKTRACK
[]            ld[i - col + N - 1] = rd[i + col] = cl[i] = 0;
[]         }
[]      }
[]      return false;
[]   }
```

2 Nhánh và cận (Branch and Bound)

2.1 Phương pháp

Xét bài toán tối ưu, chúng ta có thể thông qua phương pháp quay lui để có thể liệt kê ra các tập nghiệm có thể của bài toán, rồi qua đó chúng ta sẽ xét điều kiện để tìm ra phương án tốt nhất. Nhưng nếu xét hết tất cả các cấu hình thì sẽ mất rất nhiều thời gian và có thể lãng phí. Nếu chúng ta có thể đánh giá tại mỗi bước của thuật toán quay lui, khi đi tiếp bước tiếp theo không mang lại kết quả tốt hơn so với cấu hình tốt nhất hiện tại. Chúng ta có thể loại bỏ sớm phương án đó. Kỹ thuật này gọi là kỹ thuật đánh giá nhánh cận trong tiến trình quay lui. Mô hình của thuật toán nhánh cận có thể mô tả như sau:

```
[]   <Khởi tạo một cấu hình bất kì cho BESTANS>
[]   void backtrack(i) {
[]      for(mọi giá trị V có thể gán cho x[i]) {
[]         <thử x[i] := V>;
[]         if( việc thử trên vẫn còn hi vọng cho ra cấu hình tốt hơn BESTANS) {
[]              if( x[i] là phần tử cuối cùng trong cấu hình) {
[]              <cập nhật BESTANS>;
[]            } else {
[]              <ghi nhận việc thử x[i] = V (nếu cần)>;
[]              backtrack(i + 1);
[]              <bỏ ghi nhận việc thử x[i] := V (nếu cần)>;
[]            }
[]         }
[]      }
[]   }
```

2.2 Xét bài toán người du lịch

Chúng ta cùng xem bài toán người du lịch Cho n thành phố đánh số từ 1 đến n và m tuyến đường giao thông hai chiều giữa chúng, mạng lưới giao thông này được cho bởi bảng C cấp $n \times n$, ở đây $C[i,j] = C[j,i] =$ Chi phí đi đoạn đường trực tiếp từ thành phố i đến thành phố j. Giả thiết rằng $C[i,i] = 0$ với $\forall i$ $C[i,j] = +\infty$ nếu không có đường trực tiếp từ thành phố i đến thành phố j.
Một người du lịch xuất phát từ thành phố 1, muốn đi thăm tất cả các thành phố còn lại mỗi thành phố đúng 1 lần và cuối cùng quay lại thành phố 1. Hãy chỉ ra cho người đó hành trình với chi phí ít nhất. Bài toán đó gọi là bài toán người du lịch hay bài toán hành trình của một thương gia (Traveling Salesman).
Theo thuật toán đệ quy chúng ta sẽ xây dựng để liệt kê các lộ trình của thương gia như sau:

```
[]   void backtrack(int i) {
[]      for(int next_city = 1; next_city <= n; next_city++)
[]         if(visited[next_city] == false) {
[]            X[i] = next_city;
[]            visited[next_city] = true;
[]            Cost = Cost + C[X[i-1]][next_city];
[]            if( i == n )
```

```
[]              BestCost = min(BestCost, Cost + C[X[n]][1]);
[]           else
[]              backtrack(i+1);
[]           X[i] = 0;
[]           visited[next_city] = false;
[]           Cost = Cost - C[X[i-1]][next_city];
[]        }
[]   }
```

Ta có thể thấy rằng do chi phí di chuyển giữa hai thành phố luôn không âm nên nếu chi phí của lộ hiện tại đã lớn hơn chi phí của lộ trình tốt nhất chúng ta đã tìm được thì chúng ta không cần thiết tìm tiếp phần sau của lộ trình đang xây dựng nữa.

```
[]   void backtrack(int i) {
[]      for(int next_city = 1; next_city <= n; next_city++)
[]         if(visited[next_city] == false) {
[]            // Điều kiện nhánh cận
[]            if(cost+C[X[[i-1]][next_city] < BestCost)
[]               X[i] = next_city;
[]               visited[next_city] = true;
[]               Cost = Cost + C[X[i-1]][next_city];
[]               if( i == n )
[]                  BestCost = min(BestCost, Cost + C[X[n]][1]);
[]               else
[]                  backtrack(i+1);
[]               X[i] = 0;
[]               visited[next_city] = false;
[]               Cost = Cost - C[X[i-1]][next_city];
[]            }
[]   }
```

3 Tham lam (Greedy Method)

Phương pháp nhánh cận là cải tiến của phương pháp quay lui, phương pháp này đánh giá các nghiệm mở rộng để loại bỏ đi những phương án không cần thiết, giúp cho việc tìm nghiệm tối ưu nhanh hơn.
Tuy nhiên, không phải lúc nào chúng ta cũng có thể đánh giá được nghiệm mở rộng, hoặc nếu có đánh giá được thì số phương án cần xét vẫn rất lớn, không thể đáp ứng được trong thời gian cho phép. Khi đó, người ta chấp nhận tìm những nghiệm gần đúng so với nghiệm tối ưu. Phương pháp tham lam được sử dụng trong các trường hợp như vậy. Ưu điểm nổi bật của phương pháp tham lam là độ phức tạp nhỏ, thường nhanh chóng tìm được lời giải.

3.1 Phương pháp

Giả sử nghiệm của bài toán có thể biểu diễn dưới dạng một vector $(x_1, x_2, \ldots, x_n)$ mỗi thành phần x_i được chọn ra từ tập S_i . Mỗi nghiệm của bài toán $X = (x_1, x_2, \ldots, x_n)$ được xác định "độ tốt" bằng một hàm $f(X)$ và mục tiêu cần tìm nghiệm có giá trị $f(X)$ càng lớn càng tốt (hoặc càng nhỏ càng tốt).

Tư tưởng của phương pháp tham ăn như sau: Ta xây dựng vector nghiệm X dần từng bước, bắt đầu từ vector không ( ). Giả sử đã xây dựng được $(k-1)$ thành phần $(x_1, x_2, \ldots, x_(k-1))$ của nghiệm và khi mở rộng nghiệm ta sẽ chọn x_k "tốt nhất" trong các ứng cử viên trong tập Sk để được $(x_1, x_2, \ldots, x_k)$ Việc lựa chọn như thế được thực hiện bởi một hàm chọn. Cứ tiếp tục xây dựng, cho đến khi xây dựng xong hết thành phần của nghiệm.
Mô hình của phương pháp tham lam:

```
[]   void Greedy() {
[]      <Khởi tạo tập nghiệm rỗng X[]>;
[]      int i = 0;
[]      while( chưa xây dựng xong hết thành phần của nghiệm) {
[]         i++;
[]         <xác định S[i]>;
[]         <X[] = ứng cử viên tốt nhất trong tập S[i]>;
[]      }
[]   }
```

3.2 Bài toán ATM

Một máy ATM hiện có n $(n \leq 20)$ tờ tiền có giá trị $t_1, t_2, \ldots, t_n$. Hãy tìm cách trả ít tờ nhất với số tiền đúng bằng S.
input:
- Dòng đầu là 2 số n và S
- Dòng thứ 2 gồm n số $t_1, t_2, \ldots, t_n$
output:
Nếu có thể trả tiền đúng bằng S thì đưa ra số tờ ít nhất cần trả và đưa ra cách trả, nếu không khi -1.
Thuật toán với ý tưởng tham lam như sau:
- Sắp xếp các tờ tiền giảm dần theo giá trị
- Lần lượt xét các tờ tiền có giá trị nhỏ, nếu vẫn chưa thấy đủ S và tờ tiền đang xét có giá trị nhỏ hơn hoặc bằng S thì lấy luôn tờ tiền đó(tham lam).

```
[]   bool cmp(int a, int b) {
[]      return (a > b);
[]   }
[]   int main()
[]   {
[]      int n,s;
[]      int *a;
[]      a = new int[n]; // cấp phát động để nhập các phần tử
[]      for(int i = 0; i < n; i++)
[]         cin >> a[i];
[]      sort(a, a+n, cmp) // sắp xếp các tờ tiền giảm dần theo giá trị
[]      vector<int> ans;
[]      int dem = 0;
[]      for(int i = 0; i < n; i++)
[]         if(a[i] < s) {
[]            ans.push_back(a[i]);
[]            s -= a[i];
[]            dem++;
[]         }
```

```
[]      if( s != 0)
[]         cout<<-1;
[]      else {
[]         cout << dem << "\n";
[]         for(int i = 0; i< n; i++)
[]            cout << ans[i] << " ";
[]      }
[]   }
```

4 Chia để trị (Divide and Conquer)

4.1 Phương pháp

Chia để trị là một phương pháp áp dụng cho các bài toán có thể chia nhỏ thành các bài toán con. Các bài toán con lại được tiếp tục phân thành các bài toán con nhỏ hơn cứ thế tiếp tục cho tới khi ta nhận được bài toán có thể giải quyết được. Sau đó kết hợp lại các kết quả của bài toán con để nhận được nghiệm của bài toán con lớn hơn, để cuối cùng nhận được nghiệm của bài toán cần giải. Thông thường các bài toán con nhận được trong quá trình phân chia là cùng dạng với bài toán ban đầu, chỉ có cỡ của chúng là nhỏ hơn. Thuật toán chia để trị có thể biểu diễn bằng mô hình đệ quy như sau:

```
[]   void DivideConquer(A,x) { // tìm nghiệm x của bài toán A
[]      if(A đủ nhỏ)
[]         Solve(A);
[]      else {
[]         <Phân A thành các bài toán con A[1], A[2], ... A[m]>;
[]         for( int i=1; i<=m; i++) {
[]            DivideConquer(A[i],x[i]);
[]            <Kết hợp các nghiệm x[i] (i=1,2,..,m) của bài toán con
[]            A[i] đểnhận được nghiệm của bài toán A>;
[]         }
[]      }
[]   }
```

4.2 Bài toán tính lũy thừa

Bài toán: Cho số a và số nguyên dương n, tính a^n.
Thông thường để tính a^n thì chúng ta mất n phép tính nhân. Nhưng khi áp dụng chia để trị vào bài toán này thì sẽ khác. Ý tưởng là ta sẽ tính a^n dựa vào a^k (trong đó $k = n \div 2$) như sau:
- Nếu n chẵn: $a^n = a^k \times a^k$
- Nếu n lẻ: $a^n = a^k \times a^k \times a$
Để tính a^k ta lại dựa vào $a^{k \div 2}$, quá trình chia nhỏ cho đến khi nhận được bài toán tính a^1 thì dừng.

```
[]   long long BinPow(long long a, long long n){
[]      if(n == 1) {
[]         return a;
[]      } else {
```

```
[]          long long temp = BinPow(a, n/2);
[]          if(n % 2 == 0)
[]             return temp * temp; // n chẵn
[]          else
[]             return temp * temp * a; // n lẻ
[]       }
[]    }
```

Để đánh giá thời gian thực hiện thuật toán, ta tính số phép nhân phải sử dụng, gọi $T(n)$ là số phép nhân thực hiện, ta có:

$$\begin{cases} T(1) = 1 \\ T(n) \leq T\left(\frac{n}{2}\right) + 1 + 1 \text{ nếu } n > 1 \end{cases}$$

$$T(n) \leq T\left(\frac{n}{2}\right) + 2 \leq T\left(\frac{n}{2^2}\right) + 2 + 2 \leq \cdots \leq 2\log n$$

Như vậy, thuật toán chia trị mấy không quá $2\log n$ phép nhân, nhỏ hơn rất nhiều so với n phép nhân.