

data structure

Cấu trúc dữ liệu

and

và

algorithms

Giải thuật



A. TỔNG QUAN:

Các cấu trúc dữ liệu và các giải thuật được xem như là 2 yếu tố quan trọng nhất trong lập trình, đúng như câu nói nổi tiếng của Niklaus Wirth:

$$\begin{aligned} \text{Chương trình} &= \text{Cấu trúc dữ liệu} + \text{Giải thuật} \\ (\text{Programs}) &= (\text{Data Structures}) + (\text{Algorithms}). \end{aligned}$$

Giải thuật: Tập hợp hữu hạn các chỉ dẫn (lệnh) được xác định rõ để giải quyết vấn đề cụ thể (sort, search, DFS, BFS, ...);

Cấu trúc dữ liệu: Là tập hợp các kiểu dữ liệu được sắp xếp theo một trật tự cụ thể. Là một cách lưu trữ (tổ chức) dữ liệu trong máy tính sao cho nó có thể được sử dụng một cách hiệu quả (stack, queue, array, list, graph, tree, ...).

Một cấu trúc dữ liệu tốt sẽ giúp ta quản lý dữ liệu một cách hiệu quả, chính xác hơn; một thuật toán tốt sẽ giúp xử lý dữ liệu nhanh chóng, rõ ràng hơn.

I. Phân tích thuật toán:

1. Định nghĩa:

- Là 1 chuỗi các bước tính toán được định nghĩa rõ ràng để giải quyết 1 vấn đề;
- Nhận vào một tập các giá trị đầu vào;
- Trả về một tập các giá trị đầu ra;
- Biểu diễn bằng: mã nguồn, mã giả;
- Tính đúng đắn (cần thiết): kết quả trả về phản ánh đúng mong muốn của thông tin nhận vào;
- Tính hiệu quả (quan trọng): độ tin cậy, tốc độ xử lý, tài nguyên sử dụng.

2. Đánh giá:

- Một vấn đề giải quyết bởi thuật toán khác nhau
- Với mỗi thuật toán
 - Thời gian: thời gian chạy của thuật toán;
 - Không gian: dung lượng bộ nhớ sử dụng;
- Thời gian chạy
 - Dữ liệu đầu vào;
 - Kỹ năng lập trình;

- Chương trình dịch, hệ điều hành;
- Tốc độ phép toán trên máy tính;
- Phân tích thực nghiệm:
 - Đo thời gian chạy thực, vẽ đồ thị;
 - Cần xây dựng kịch bản thực nghiệm, cài đặt thuật toán;
 - Để so sánh các thuật toán, phải sử dụng cùng môi trường;
 - Phù hợp cho dự đoán.
- Phân tích toán học:
 - Ước lượng số phép toán, tính độ phức tạp (BigO notation);
 - Chỉ cần xây dựng mã giả, không cần cài đặt thuật toán;
 - Để so sánh các thuật toán, không cần cùng môi trường;
 - Phù hợp cho dự đoán và giải thích.
- Trường hợp xấu nhất (thông thường)
 - Thời gian chạy lớn nhất của thuật toán trên tất cả các dữ liệu có cùng độ lớn.
- Trường hợp trung bình (đôi khi)
 - Thời gian chạy trung bình của thuật toán trên tất cả các dữ liệu có cùng độ lớn.
 - Khó: do phải biết phân phối xác suất của dữ liệu
- Trường hợp tốt nhất (hiếm):
 - Thời gian chạy ít nhất của thuật toán trên tất cả các dữ liệu có cùng độ lớn.

3. Mã giả (pseudo – code):

- Mô tả bậc cao của một thuật toán;
- Cấu trúc rõ ràng hơn văn xuôi;
- Không chi tiết như mã nguồn;
- Được ưa thích trong biểu diễn giải thuật;
- Ân đi các khía cạnh thiết kế chương trình.

4. Phân tích độ phức tạp thời gian:

<https://howteam.vn/course/cau-truc-du-lieu-va-giai-thuat/do-phuc-tap-thoi-gian-bigo-la-gi-4270>

- Đánh giá thời gian chạy của thuật toán thông qua $T(n)$ (số lần phép toán sơ cấp cần thực hiện: số học, logic, so sánh,...). Chỉ quan tâm đến tốc độ tăng của hàm.
- *Đối với lập trình thi đấu, thường khi ta nói “một thuật toán có độ phức tạp $O(A)$ ” có nghĩa là có tối đa A câu lệnh $O(1)$ được thực thi. Việc đánh giá độ phức tạp chính là tính số lần thực thi tối đa của một câu lệnh $O(1)$.*
- Độ tăng của hàm:
 - + Với n là độ lớn dữ liệu đầu vào:
 - + Tỷ lệ tăng trưởng (chính xác):
 - $an^2 + bn + c$
 - $an + b$
 - $an \log n + bn + c$
 - + Bậc tăng trưởng (xấp xỉ):
 - $an^2 + bn + c \rightarrow$ bậc n^2 ;
 - $an + b \rightarrow$ bậc n ;
 - $an \log n + bn + c \rightarrow$ bậc $n \log(n)$

- Các độ phức tạp thuật toán thường gặp trong ví dụ cụ thể:

- + Sequential Statements: (Statement with basic operations)
 - Comparisons
 - Assignments
 - Reading a variable
 - ...

⇒ Độ phức tạp thời gian mỗi câu lệnh như vậy là: $O(1)$

⇒ Tổng độ phức tạp thời gian vẫn là $O(1)$.

⇒ Tổng thời gian chạy của sequential statements:

$$T(n) = t(statement1) + t(statement2) + \dots + t(statementN)$$

- Conditional Statements:

- Giả sử rằng ta có một đoạn code:

```
1  if (isValid) {  
2      statement1;  
3      statement2;  
4  } else {  
5      statement3;  
6  }
```

- Thì độ phức tạp thời gian là:

$$T(n) = \text{Math.max}([t(statement1) + t(statement2)], t(statement3)])$$

- Và độ phức tạp thuật toán cũng được tính theo cách tương tự:
Nếu trong block if có độ phức tạp $O(n\log n)$ và block else có độ phức tạp $O(1)$. Thì tổng độ phức tạp của thuật toán là $O(n\log n)$.

- Loop Statements: ($n = \text{array.length}$)

```
1  for (let i = 0; i < array.length; i++) {  
2      statement1;  
3      statement2;  
4  }
```

- Độ phức tạp thời gian:

$$T(n) = n \times [t(statement1) + t(statement2)]$$

- Độ phức tạp thuật toán: = số lần lặp = $O(n)$
 - Tuy nhiên khi $n = \text{constant}$ (ví dụ: $n = 1, 4, 500, \dots$) thì độ phức tạp của thuật toán là $O(1), O(4), O(500), \dots$ đều quy về $O(1)$.

- Nested Loop Statements:

```
1  for (let i = 0; i < n; i++) {  
2      statement1;  
3  
4      for (let j = 0; j < m; j++) {  
5          statement2;  
6          statement3;  
7      }  
8  }
```

- Độ phức tạp thời gian:

$$T(n) = n \times \{ t(statement1) + m \times [t(statement2) + t(statement3)] \}$$

- Độ phức tạp thuận toán: $O(m^*n)$ tương đương với $O(n^2)$.

- Binary search:

- Giả sử đề bài yêu cầu tìm kiếm số trong một dãy đã được sắp xếp, áp dụng thuật toán chia để trị bằng cách gán dãy vào một mảng có 8 phần tử (giả sử).
- Khi đó độ lớn dữ liệu đầu vào là $n = 8$, ta thực hiện phép chia đôi (chia cho 2) cho đến khi nhận được kết quả là 1 (nhận được 1 số duy nhất và cũng chính là kết quả cần tìm).
- Cụ thể: $8/2/2/2 = 1$, khi đó dừng thuật toán và cho ra kết quả.
- Ta có biến đổi biểu thức trên như sau:

$$\begin{aligned} 8 &= 2^3 \quad (3_la_so_lan_chia_va_cung_la_do_phuc_tap_O(n)) \\ \Leftrightarrow n &= 2^{O(n)} \\ \Leftrightarrow O(n) &= \log_2(n) \\ \Rightarrow O(\log n) & \end{aligned}$$

TÊN	ĐỊNH NGHĨA	VÍ DỤ
Độ phức tạp hằng số $O(1)$	Một thuật toán được gọi là có độ phức tạp hằng số $O(1)$ khi thời gian chạy không phụ thuộc vào kích thước dữ liệu đầu vào.	các toán tử cơ bản, truy cập vào một phần tử của mảng, in ra 1 xâu, ...
Độ phức tạp tuyến tính $O(n)$	Một thuật toán được gọi là có độ phức tạp tuyến tính $O(n)$ khi thời gian chạy thay đổi tuyến tính phụ thuộc vào kích thước dữ liệu đầu vào.	Kích thước dữ liệu đầu vào là 10, thời gian chạy mất 1s. Khi kích thước dữ liệu đầu vào là 100, thời gian chạy mất 10s thì khi đó ta nói thuật toán có độ phức tạp $O(n)$.
Độ phức tạp theo hàm số logarit $O(\log n)$	Một thuật toán được gọi là có độ phức tạp theo hàm số logarit $O(\log n)$ khi thời gian chạy thay đổi theo dạng hàm số logarit phụ thuộc vào kích thước dữ liệu đầu vào.	tìm kiếm nhị phân, các thao tác với cây nhị phân, ...

Chú ý:

- Khi ta nói độ phức tạp của thuật toán là $O(\log n)$ thì $\log n$ ở đây không phải là $\log_{10} n$ như trong Toán học. Trên thực tế, với tất cả các hàm logarit có cơ số lớn hơn 1, chúng đều tiệm cận nhau, do đó chúng ta sẽ nói chung là $\log n$ kể cả khi khác cơ số.
- Sẽ có những lúc tồn tại hàm $O(k * n)$ với hằng số k. Thực chất, viết như vậy ta vẫn sẽ hiểu độ phức tạp tăng tuyến tính với n. Việc viết thêm k vào giúp ta có đánh giá chính xác hơn về độ phức tạp thuật toán. Tuy nhiên, ta chỉ quan tâm đến hằng số k khi k thật sự có một ảnh hưởng lớn lên độ phức tạp bài toán. Với đa số thuật toán, k là **không đáng kể**.
- Khi so sánh các thuật toán bằng cách đánh giá độ phức tạp thời gian BigO thì kết quả sẽ chỉ là **tương đối**. Hai thuật toán có cùng độ phức tạp có thể có số câu lệnh cơ bản khác nhau dẫn đến thời gian chạy sẽ khác nhau (Lí do là do hằng số k được nêu ở trên).

❖ Độ phức tạp (thời gian) trong bài toán đệ quy (recursive):

- ✓ <https://chuyentinhparis.com/tinh-do-phuc-tap-cua-thuat-toan-de-quy/>
- ✓ <https://vnoi.info/wiki/translate/topcoder/Computational-Complexity-Section-2>
- ✓ Nên xem: <https://tek4.vn/khoa-hoc/cau-truc-du-lieu-va-thuat-toan/uoc-luong-do-phuc-tap-cua-de-quy-bang-dinh-ly-tho-master-theorem>

- *Nguyên tắc*: Dựa vào định nghĩa, để giải quyết vấn đề trên ta tính số lần các câu lệnh O(1) được thực hiện qua mỗi lần đệ quy, sau đó tính tổng. Tổng vừa tìm được chính là độ phức tạp (thời gian) của đệ quy.
- *Lưu ý*: Tổng nói ở trên mới là độ phức tạp của đệ quy khi các câu lệnh còn lại đồng độ phức tạp O(1), khi các câu lệnh còn lại khác O(1) thì tính tích của (tổng độ phức tạp lệnh đệ quy) và (độ phức tạp các câu lệnh đó). Xem ví dụ cuối cùng – ví dụ 5.
- *Ví dụ*:

- 1)

```
int recursiveFun1 (int n)
{
    if (n <= 0)
        return 1;
    else
        return 1 + recursiveFun1 (n - 1);
}
```

- Độ phức tạp là $O(n)$

- Lí do:

- + Với dữ liệu đầu vào $n > 0$, hàm recursiveFun1 sẽ đệ quy theo thông số giảm dần ($n - 1$) cho đến khi $n = 0$. Có thể hiểu n chạy n + 1 lần: từ n đến 0; (cộng 1 cho lần return khi $n = 0$)

- + Độ phức tạp của thuật toán trên gồm 2 phần: Độ phức tạp lệnh return + kiểm tra điều kiện và Độ phức tạp lệnh đệ quy. Trong đó Độ phức tạp của lệnh return + kiểm tra điều kiện là $O(1)$ nên độ phức tạp của thuật toán là Độ phức tạp lệnh đệ quy nhân với $O(1)$ và cũng chính là Độ phức tạp lệnh đệ quy.
- + Giả sử $n = 3$:
 - recursiveFun1 (3)
 - recursiveFun1 (2)
 - recursiveFun1 (1)
 - recursiveFun1 (0) → dừng.
- + Qua đó ta có nhận xét độ phức tạp lệnh đệ quy qua $n + 1$ lần chạy là $n + 1$ do có $n + 1$ lần các câu lệnh $O(1)$ được thực thi, tức $O(n + 1)$ hay $O(n)$. Đây được gọi là độ phức tạp tuyến tính.

o 2)

```
int recursiveFun2 (int n)
{
    if (n <= 0)
        return 1;
    else
        return 1 + recursiveFun2 (n - 5);
}
```

- Độ phức tạp là $O(n)$
- Lí do:
 - + Với dữ liệu đầu vào $n > 0$, hàm recursiveFun2 sẽ đệ quy theo thông số giảm dần ($n - 5$) cho đến khi $n \leq 0$. Có thể hiểu n chạy $n/5+1$ lần (làm tròn lên): từ n đến 0 theo cách giảm 5 đơn vị mỗi lần đệ quy; (cộng 1 cho lần return khi $n = 0$)
 - + Độ phức tạp của thuật toán trên gồm 2 phần: Độ phức tạp lệnh return + kiểm tra điều kiện và Độ phức tạp lệnh đệ quy. Trong đó Độ phức tạp của lệnh return + kiểm tra điều kiện là $O(1)$ nên độ phức tạp của thuật toán là Độ phức tạp lệnh đệ quy nhân với $O(1)$ và cũng chính là Độ phức tạp lệnh đệ quy.
 - + Giả sử $n = 15$:

- recursiveFun2 (15)
 - recursiveFun2 (10)
 - recursiveFun2 (5)
 - recursiveFun2 (0) → dừng.
- + Qua đó ta có nhận xét độ phức tạp lệnh đệ quy qua $(n/5 + 1)$ lần chạy là $(n/5 + 1)$ do có $(n/5 + 1)$ lần các câu lệnh O(1) được thực thi, tức $O((n/5 + 1))$ hay $O(n/5)$ hay $O(n)$. Đây cũng được gọi là độ phức tạp tuyến tính.
- 3)


```
int recursiveFun3 (int n)
{
    if (n <= 0)
        return 1;
    else
        return 1 + recursiveFun3 (n/5);
}
```
- Độ phức tạp là $O(\log n)$ (cơ số 5, tuy nhiên vì các hàm log cơ số dương phân biệt đều tiệm cận với nhau nên có thể xem chung là $\log n$, trong lập trình ta thường hiểu $\log n$ là \log cơ số 2 của n). (cộng 1 cho lần return khi $n = 0$)
- Lý do:
 - + Với dữ liệu đầu vào $n > 0$, hàm recursiveFun3 sẽ đệ quy theo thông số giảm dần ($n/5$) cho đến khi $n = 0$. Có thể hiểu n chạy $\log_5(n) + 1$ lần (làm tròn lên): từ n đến 0 theo cách giảm 5 lần mỗi lần đệ quy (giải thích tại + thứ 4).
 - + Độ phức tạp của thuật toán trên gồm 2 phần: Độ phức tạp lệnh return + kiểm tra điều kiện và Độ phức tạp lệnh đệ quy. Trong đó Độ phức tạp của lệnh return + kiểm tra điều kiện là $O(1)$ nên độ phức tạp của thuật toán là Độ phức tạp lệnh đệ quy nhân với $O(1)$ và cũng chính là Độ phức tạp lệnh đệ quy.
 - + Giả sử $n = 20$:
 - recursiveFun3 (20)
 - recursiveFun3 (4)

- recursiveFun3 (0) → dừng.
- + Gọi m là độ phức tạp của thuật toán (số lần mà câu lệnh O(1) được thực thi). Khi đó ta cần thực hiện m lần đệ quy để kết thúc thuật toán, để tính m, ta tính số lần chia 20 cho 5 cho đến khi thu được kết quả là 1. Cụ thể: $20/5/5/..../5 = 1$ hay $20/5^m = 1$ hay $20 = 5^m$ hay $m = \log_5(20)$. Cộng thêm 1 lần chạy O(1) câu lệnh return + kiểm tra điều kiện dừng cuối cùng khi $n = 0$ ta được độ phức tạp = $\log_5(20) + 1$.
- + Qua đó ta có nhận xét độ phức tạp lệnh đệ quy qua $\log_5(n) + 1$ lần chạy là $\log_5(n) + 1$ do có $\log_5(n) + 1$ lần các câu lệnh O(1) được thực thi, tức $O(\log_5(n) + 1)$ hay $O(\log_5(n))$ hay $O(\log n)$. Đây được gọi là độ phức tạp logarit.

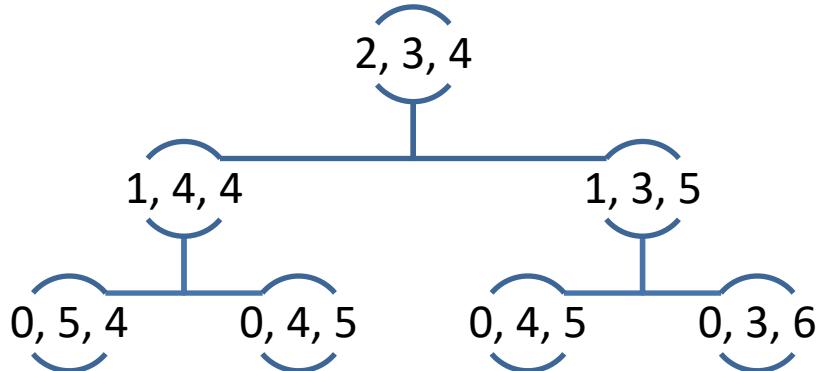
o 4)

```
void recursiveFun4(int n, int m, int o)
{
    if (n <= 0)
    {
        printf("%d, %d\n", m, o);
    }
    else
    {
        recursiveFun4(n-1, m+1, o);
        recursiveFun4(n-1, m, o+1);
    }
}
```

- Độ phức tạp là $O(2^n)$
- Lí do:
 - + Với dữ liệu đầu vào $n > 0$, hàm recursiveFun4 sẽ đệ quy từng lần một, tương tự như tìm kiếm theo chiều sâu Depth – First Search. Có thể hiểu n chạy $2^n + 1$ lần (2 là số hàm đệ quy liên tục) (giải thích sau)
 - + Độ phức tạp của thuật toán trên gồm 2 phần: Độ phức tạp lệnh print + kiểm tra điều kiện và Độ phức tạp lệnh đệ quy. Trong đó Độ phức tạp của lệnh print + kiểm tra điều kiện là $O(1)$ nên độ phức tạp của thuật

toán là Độ phức tạp lệnh đệ quy nhân với $O(1)$ và cũng chính là Độ phức tạp lệnh đệ quy.

- + Giả sử $n = 2, m = 3, o = 4$ cách chạy đệ quy được mô phỏng bởi sơ đồ bên dưới:



- + Nhận xét: Sơ đồ có dạng một full – 2 ary tree hay còn gọi là complete binary tree (cây nhị phân đầy đủ) có chiều cao (height) là $n = 2$. Số kết quả thu được chính là số lá của full – 2 ary tree, theo công thức ta có số lá bằng $m^h = 2^2 = 4$.
- + Tổng quát với một giải thuật với m câu lệnh đệ quy liên tiếp, bài toán đưa về yêu cầu tìm số lá của một full – m ary tree có độ cao height là n ($h = n$). Từ đó suy ra số lá = $m^h = m^n$. Đó cũng chính là độ phức tạp của giải thuật đệ quy m câu lệnh liên tiếp nhau.
- + Qua đó ta có nhận xét độ phức tạp lệnh đệ quy m câu lệnh liên tiếp là m^n , do có m^n lần các câu lệnh $O(1)$ được thực thi, tức $O(m^n)$. Đây được gọi là độ phức tạp mũ.

o 5)

```

int recursiveFun5(int n)
{
    for (i = 0; i < n; i += 2) {
        // do something
    }

    if (n <= 0)
        return 1;
    else
  
```

```
        return 1 + recursiveFun5(n-5);  
    }
```

- Độ phức tạp $O(n^2)$
- Lý do:
 - + Với dữ liệu đầu vào $n > 0$, hàm recursiveFun5 sẽ đệ quy theo thông số giảm dần ($n - 5$) cho đến khi $n \leq 0$. Có thể hiểu n chạy $n/5+1$ lần (lần tròn lên): từ n đến 0 theo cách giảm 5 đơn vị mỗi lần đệ quy; (cộng 1 cho lần return khi $n = 0$).
 - + Độ phức tạp của thuật toán gồm 3 phần: Độ phức tạp lệnh return + kiểm tra điều kiện và Độ phức tạp lệnh đệ quy và Độ phức tạp vòng lặp for. Trong đó Độ phức tạp của lệnh return + kiểm tra điều kiện là $O(1)$ và Độ phức tạp của vòng lặp for là $O(n)$ nên độ phức tạp của thuật toán là Độ phức tạp lệnh đệ quy nhân với $O(1)$ nhân với $O(n)$ và cũng chính là Độ phức tạp lệnh đệ quy nhân với $O(n)$.
 - + Mà độ phức tạp đệ quy được giải thích trong ví dụ 2. Vậy độ phức tạp cả thuật toán là $O(n^2)$.

- Ký hiệu tiệm cận (O):

▶ ▶ **O - Ô-lớn (chặn trên):**

Ta nói rằng $f(n)$ là ô-lớn của $g(n)$ nếu tồn tại các hằng số $c > 0$, và $n_0 > 0$ sao cho:

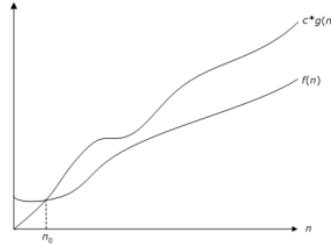
$$0 \leq f(n) \leq cg(n)$$

với mọi $n \geq n_0$

Ký hiệu: $f(n) \in O(g(n))$

▶ Ví dụ: $2n^2 \in O(n^3)$

Hàm không phải giá trị



Ý nghĩa: $g(n)$ là tiệm cận trên của $f(n)$.

Nguyên văn tiếng anh: $O(g(n))$ is the set of functions with smaller or the same order of growth as $g(n)$. ~ một tập hợp các hàm mà có bậc tăng trưởng nhỏ hơn hoặc bằng với $g(n)$

Ví dụ: $O(n^2)$ bao gồm $O(1)$, $O(n)$, $O(n\log n)$, ...

Big-O Examples

Example-1: Tìm upper bound của hàm $f(n) = 3n+8$

Solution: Ta đã có định nghĩa: $O(g(n)) = \{f(n): \text{tồn tại các hằng số dương } c \text{ và } n_0 \text{ sao cho } 0 \leq f(n) \leq c * g(n) \text{ với mọi } n \geq n_0\}.$ *

=> Để tìm upper bound đồng nghĩa với ta phải **tìm được hằng số dương c và n_0** thỏa mãn điều kiện $0 \leq f(n) \leq c * g(n)$ với mọi $n \geq n_0$

Với bài toán đã cho: $3n+8 \leq 4n$ với mọi $n \geq 8$

=> Ta có upper bound $O(n)$ với $c = 4$ và $n_0 = 8$.

Example-2: Tìm upper bound của hàm $f(n) = n^2 + 1$

Solution: Tương tự example 1, ta đi tìm c và n_0

Ta có $n^2 + 1 \leq 2 * n^2$ với mọi $n \geq 1$

=> upper bound $O(n^2)$ với $c = 2$ và $n_0 = 1$

Example-3: Tìm upper bound của hàm $f(n) = n^4 + 100 * n^2 + 50$

Solution: Ta có $n^4 + 100 * n^2 + 50 \leq 2 * n^4$ với mọi $n \geq 11$

=> upper bound $O(n^4)$ với $c = 2$ và $n_0 = 11$

(Các bất đẳng thức này chỉ đơn giản là chuyển về cộng trừ bất đẳng thức rồi khai căn khá đơn giản nên mình sẽ không trình bày lại chi tiết)

5. Phân tích tiệm cận Thuật toán:

- Xác định thời gian chạy sử dụng kí hiệu O
- + Tìm số lần các phép toán sơ cấp thực hiện nhiều nhất dưới dạng hàm số với biến là độ lớn dữ liệu đầu vào n ($f(n)$);
- + Miêu tả hàm này theo kí hiệu O.
- Ví dụ:
 - + Thuật toán arraySum (A, n) chạy nhiều nhất $(4n + 2)$ phép toán sơ cấp;
 - + Ta nói thuật toán arraySum (A, n) chạy trong thời gian $O(n)$.

6. Kỹ thuật đánh giá thời gian chạy:

- Thời gian chạy **lệnh gán**: = thời gian thực hiện **biểu thức**
- Thời gian chạy **lệnh lựa chọn**: = thời gian thực hiện **điều kiện + min** (thời gian thực hiện **lệnh khối if**, thời gian thực hiện **lệnh khối else**)
- Thời gian chạy **lệnh lặp**: = **tổng** thời gian chạy **từng vòng lặp** (Trong đó: thời gian chạy từng vòng lặp = thời gian thực hiện điều kiện + thời gian thực hiện nội dung vòng lặp đang xét)

► **Lệnh lặp: for, while, do-while**

$$\sum_{i=1}^{X(n)} (T_0(n) + T_i(n))$$

$X(n)$: số vòng lặp

$T_0(n)$: điều kiện lặp

$T_i(n)$: thời gian thực hiện vòng lặp thứ i

II. Các dạng giải thuật (Algorithm Paradigms):

Nội dung:

1. Vét cạn – Complete Search / Brute – force Search;
2. Chia để trị - Divide and Conquer;
3. Ngày thơ – Naive;
4. Tham lam – Greedy;
5. Quy hoạch động – Dynamic Programming;

I. Giải thuật vét cạn (brute – force):

1. Tổng quan:

Vét cạn mang nặng tinh thần của phương pháp BackTracking (quay lui) (Xem [phương pháp BackTracking](#)) hầu hết các bài toán sử dụng vét cạn đều dùng tới backTracking để giải quyết, và khi đó bài toán được giải cũng sẽ là những vấn đề rất khó vì khi dùng đến backTrack nghĩa là không còn thuật toán nào tối ưu hơn để giải, phải xét tất cả các khả năng có thể xảy ra để đưa ra kết luận.

2. Các dạng yêu cầu dùng vét cạn:

- a. Bài toán với dữ liệu đầu vào không quá lớn (hoặc đã biết trước và hữu hạn): để đảm bảo thời gian chạy nhanh, thuật toán nằm trong tầm kiểm soát;
- b. Bài toán sinh ra các cấu hình có cách hình thành giống nhau (như sinh hoán vị, sinh chuỗi bit,...) → backTracking.
- c. Bài toán khi áp dụng tất cả các giải thuật khác không ra kết quả (hoặc khi nhìn vào ra ngay cách làm bằng vét cạn).

3. Các ví dụ cơ bản:

Ví dụ 1:

Tìm vị trí số x trên dãy a gồm N số thực.

- Input: dãy (a, N) – dãy gồm N số thực, số x – số cần tìm;
- Output: số nguyên – vị trí của x trên a (-1 nếu a không có x).

Lời giải:

```
for i = 0 : N - 1;  
if (a[i] == x) return i  
return -1
```

```
int findIndexNumber (int *ptr, int N, const int value)  
{  
    for (int i = 0; i < N; i++) {  
        if (*(ptr + i) == value) return i;  
        else continue;  
    }  
    return -1;  
}
```

Nhận xét:

- Độ phức tạp: $O(n)$

Ví dụ 2:

Cho 2 chuỗi T và chuỗi S ($20 < |S|, S < 100000$). Tìm tất cả các vị trí mà chuỗi T xuất hiện trong chuỗi P . Ví dụ minh họa:

- + Chuỗi S = “abc”
- + Chuỗi T = “abcdefghijklmabcbchgjak”
- => Chuỗi S sẽ xuất hiện trong T tại 2 vị trí là 1 và 16.

Lời giải:

Ý tưởng:

- Dùng 1 vòng lặp for xét tất cả vị trí trong S ;

- Nếu tại vị trí i của S mà $S[i]$ giống với kí tự đầu của $S1$, thì dùng 1 vòng lặp for khác để tiếp xem S và $S1$ có giống nhau không.
- Xuất ra vị trí nếu có.

```

void findIndex(const string s, string s1)
{
    for (int i = 0; i < s.length(); i++)
    {
        bool check = true; // Kiem tra s1 co nam trong s hay khong
        if (s[i] == s1[0])
        {
            for (int j = 0, k = i; j < s1.length(); j++, k++)
            {
                if (s[k] != s1[j]) // Neu khong co k thi dung i
                {
                    // -> i se bi thay doi -> sai o vong lap ngoai
                    check = false;
                    break;
                }
            }
            if (check)
                cout << i << " "; // Chi in ra vi tri dau tien cua s1 trong s
        }
    }
}

int main()
{
    const string s = "abcfoaehrbjsouabctetryabcesrdtfyjnertehtwrabcabc";
    string s1 = "abc";
    findIndex(s, s1);
    return 0;
}

```

Nhận xét:

- Độ phức tạp: Độ phức tạp $O(m*n)$ với n là độ dài chuỗi s , m là độ dài chuỗi $s1 \rightarrow O(n^2)$.

Ví dụ 3:

Cho 2 mảng số nguyên a , b có tối đa 1000000 phần tử. Tìm tất cả các cặp phần tử (i, j) sao cho $a_i = b_j = 1$. Nếu không có cặp nào thì xuất $(-1, -1)$.

=> Nếu bài này các bạn giải theo Brute Force thì sẽ chạy 1 vòng lặp tất cả phần tử của a , với mỗi phần tử của a có giá trị là 1 thì cần chạy duyệt tất cả phần tử của b xem b có phần tử nào có giá trị là 1 thì xuất ra cặp (i, j)

Lời giải:

```
int main()
{
    int *a = new int [1000000] {24, 12, 54, 65, 2, 43, 54, 1, 5};
    int *b = new int [1000000] {12, 53, 2, 54, 1, 4, 6, 5};
    bool check = false;
//Dem so luong phan tu trong 2 mang a, b
    int counta = 0, countb = 0;
    for (int i = 0; a[i]; i++) {
        counta++;
    }
    for (int i = 0; b[i]; i++) {
        countb++;
    }
//Vong lap -> brute force
    for (int i = 0; i < counta; i++) {
        for (int j = 0; j < countb; j++) {
            if (*(a + i) == *(b + j)) {
                check = true;
                cout << "(" << i << ", " << j << ")" << endl;
            }
        }
    }
    if (!check) cout << "(-1, -1)";
//Giai phong bo nho a, b trong HEAP
    delete[] a;
    delete[] b;

    return 0;
}
```

Nhận xét:

- Độ phức tạp: $O(m + n + m*n) = O(n^2)$ với n, m lần lượt là số phần tử của 2 mảng a và b .

Ví dụ 4:

Tìm cặp nghịch thế trong một mảng chưa được sắp xếp.

Giả sử:

`int arr[4] = {5, 6, 2, 3};`

thì cặp nghịch thế có dạng (ai, aj) với $(i < j)$ và $(ai > aj)$: $(5, 2), (5, 3), (6, 2), (6, 3)$.

Lời giải:

Ý tưởng: Áp dụng giải thuật vét cạn, ta cần dùng 2 vòng lặp for duyệt qua từng phần tử như ví dụ 4.

Code C++:

```
void findCNT (int *a, const int N)
{
    vector <int> tmpVector;
    //Thuật toán thể hiện từ đây
    for (int i = 0; i < N; i++) {
        for (int j = i + 1; j < N; j++) {
            if (*(a + i) >= *(a + j)) {
                tmpVector.push_back(*(a + i)), tmpVector.push_back(*(a + j));
            }
        }
    }
    //Thuật toán kết thúc ở đây
    for (int i = 0; i < tmpVector.size() - 1; i += 2) {
        cout << "(" << tmpVector[i] << ", " << tmpVector[i + 1] << ")";
        cout << "; ";
    }
}
```

Hoặc:

```
void findCNT (int *a, const int N)
{
    int *tmpArr = new int [N];
    int count = 0;
    //Thuật toán thể hiện ở đây
    for (int i = 0; i < N; i++) {
        for (int j = i + 1; j < N; j++) {
            if (*(a + i) >= *(a + j)) {
                *(tmpArr + count++) = *(a + i);
                *(tmpArr + count++) = *(a + j);
            }
        }
    }
}
```

```

//Thuật toán kết thúc ở đây
for (int i = 0; i < count - 1; i += 2) {
    cout << "(" << tmpArr[i] << ", " << tmpArr[i + 1] << ")";
    cout << "; ";
}
delete[] tmpArr;
}

```

Độ phức tạp: $O(n^2)$

Tham khảo bài toán tương tự với giải thuật tốt hơn khi độ phức tạp chỉ còn $O(n \log n)$ do dùng phương pháp chia để trị, tại đây.

Ví dụ 3:

Bài toán tìm tập con có tích lớn nhất của mảng có N phần tử.

Lời giải:

Ý tưởng: Áp dụng giải thuật vét cạn (Brute – force) với 2 vòng lặp for duyệt qua tất cả tập hợp con của mảng, nói đúng hơn là tính tích của các tập số có số phần tử tăng dần, qua mỗi lần tích tích, cần kiểm tra xem tích đó có lớn nhất hay không, nếu có thì lưu tích đó vào 1 biến nào đó và xét cho đến khi hết mảng, kết quả trả về là tích lớn nhất được lưu trong biến đó.

Độ phức tạp: $O(n^2)$.

Code C++:

```

#include <iostream>
#include <vector>
#include <utility>

using namespace std;

pair<vector<int>, int> maxProduct_SubSet(vector<int> a)
{
    pair<vector<int>, int> res; // Giá trị trả về của hàm

```

```

vector<int> sub{a[0]};           // Giá trị trả về thứ 1 trong pair --> vector
chứa tập con cần tìm
    int maxProduct = a[0];        // Giá trị trả về thứ 2 trong pair --> số
nguyên

    int N = a.size();
    for (int i = 0; i < N; i++)
    {
        int tmp = a[i];
        for (int j = i + 1; j < N; j++)
        {
            tmp *= a[j];
            if (maxProduct < tmp)
            {
                maxProduct = tmp;
                sub.assign(a.begin() + i, a.begin() + j + 1); // Sao chép tập
con tìm được cho sub
            }
        }
    }

    res.first = sub;             // Gán vector tìm được cho phần 1 trong pair
    res.second = maxProduct; // Gán giá trị tìm được cho phần 2 trong pair
    return res;
}

int main()
{
    int n;
    cin >> n;

    vector<int> a;
    for (int i = 0; i < n; i++)
    {
        int tmp;
        cin >> tmp;
        a.push_back(tmp);
    }

    pair<vector<int>, int> ans = maxProduct_SubSet(a);
    cout << "Subset: ";
    for (int i : ans.first)
    {
        cout << i << " ";
    }
    cout << endl;
    cout << "Max Product: " << ans.second;
}

```

Lưu ý: Với yêu cầu bài toán chỉ trả về maxProduct (tức không cần trả về tập con ứng với maxProduct đó), ta có cách giải khác ở thuật toán Tham lam), khi đó độ phức tạp chỉ còn $O(N)$ với N là số lượng phần tử trong mảng ban đầu. Tham khảo tại [đây](#).

Ví dụ 3:

Bài toán sinh ra tất cả tập con của N phần tử.

Lời giải:

Ý tưởng: Áp dụng giải thuật vét cạn (có thể xem như backTracking), ta sử dụng đệ quy để sinh ra tất cả các cấu hình.

Độ phức tạp: $O(2^N)$ với N là số phần tử mảng ban đầu, vì sinh ra 2^N tập con.

Code C++:

```
#include <iostream>
#include <vector>

using namespace std;

void print(vector<int> a)
{
    for (int i : a) {
        cout << i << " ";
    }
    cout << endl;
}

void generateSubSets(vector<int> &arr, vector<int> &subSet, int index)
{
    int N = arr.size();

    if (index == N) {
        print(subSet);
        return;
    }
    else {
        subSet.push_back(arr[index]);
        generateSubSets(arr, subSet, index + 1);
        subSet.pop_back();
    }
}
```

```
        generateSubSets(arr, subSet, index + 1);
    }
}

int main()
{
    const int N = 4;
    vector <int> arr {1, 2, 4, 3};
    vector <int> subSet;
    generateSubSets(arr, subSet, 0);
}
```

II. Giải thuật Ngây thơ (naive):

1. *Tổng quan:* (Nguồn chatGPT – 4 + chỉnh sửa);

Giải thuật Ngây thơ (naive) và giải thuật vét cạn (brute force) là hai thuật toán khác nhau, nhưng đôi khi được sử dụng để chỉ cùng một phương pháp giải quyết bài toán.

- **Giải thuật Ngây thơ (naive)** là một phương pháp đơn giản, thường được thực hiện thông qua việc thực hiện các vòng lặp và kiểm tra tất cả các trường hợp có thể (mỗi khả năng là 1 quy trình con). Nó không tận dụng bất kỳ thuật toán tối ưu nào và thường có độ phức tạp thời gian cao, đặc biệt khi kích thước dữ liệu tăng lên. *Ví dụ: tìm kiếm tuyến tính, sắp xếp nổi bọt.*
- **Giải thuật vét cạn (brute force)** cũng là một phương pháp đơn giản, trong đó ta kiểm tra tất cả các khả năng (mỗi khả năng là một kết quả) và lựa chọn kết quả tốt nhất. Phương pháp này không tận dụng bất kỳ thông tin cụ thể nào về bài toán và thường có độ phức tạp thời gian cao. *Ví dụ: tìm kiếm tuyến tính, liệt kê tất cả các tổ hợp.*

Tuy hai thuật toán này có thể có một số điểm tương đồng, nhưng chúng không hoàn toàn giống nhau. Thuật toán vét cạn thường chỉ áp dụng cho các bài toán mà không có cấu trúc hoặc có số lượng khả năng giới hạn (thường yêu cầu tìm một kết quả cụ thể + tốt nhất để trả về), trong khi thuật toán Ngây thơ có thể được áp dụng trong các tình huống khác nhau (thường không yêu cầu tìm kết quả tốt nhất mà là xét qua tất cả các bước để tìm ra kết quả).

Tóm lại, giải thuật Ngây thơ (naive) và giải thuật vét cạn (brute force) là hai phương pháp khác nhau trong thiết kế và thực hiện giải thuật, mặc dù có thể được sử dụng để chỉ cùng một phương pháp giải quyết bài toán trong một số trường hợp.

2. *Các ví dụ cơ bản:*

Ví dụ 1:

Bài toán Sort (sắp xếp các phần tử) của một mảng theo chiều tăng (giảm) dần.

Lời giải:

Ý tưởng: Áp dụng giải thuật vét cạn, ta cần duyệt qua tất cả phần tử của mảng, với mỗi phần tử được duyệt, duyệt tiếp các phần tử còn lại, phần tử nào nhỏ hơn sẽ được chuyển về phía trước (hoán đổi vị trí 2 phần tử đang xét ứng với 2 vòng lặp for lồng nhau).

Code C++:

```
void normalSort (int *a, const int N)
{
    for (int i = 0; i < N; i++) {
        for (int j = i + 1; j < N; j++) {
            if (*(a + i) >= *(a + j))
            {
                swap (*(a + i), *(a + j));
            }
        }
    }
}
```

Độ phức tạp: $O(n^2)$.

Tham khảo thuật toán chia để trị (devide and conquer) để có được giải thuật tốt hơn với độ phức tạp $O(n \log n)$ tại [đây](#):

Ví dụ 2:

Bài toán nhân 2 ma trận A kích cỡ ($row1, col1$) và ma trận B kích cỡ ($row2, col2$).

Lời giải:

Ý tưởng: Áp dụng giải thuật ngây thơ và dựa vào cách nhân 2 ma trận bằng tay để tìm ra quy luật và biểu diễn bằng các vòng lặp for lồng nhau. A và B phải có kích thước lần lượt là (m, p) và (p, n) , nếu không ta không thực hiện phép nhân 2 ma trận được.

Độ phức tạp: $O(mnp)$ hoặc $O(n^3)$ <cho trường hợp 2 ma trận vuông>

Pseudo – code:

Input: Hai ma trận A kích thước $n \times m$ và B kích thước $m \times p$

1: Khởi tạo ma trận C có kích thước $n \times p$

2: For i từ 1 → n :

3: For j từ 1 → p :

4: Gán $sum = 0$

5: For r từ 1 → m :

6: Gán $sum = sum + A_{i,r} \times B_{r,j}$

7: Gán $C_{i,j} = sum$

Output: Ma trận C kích thước $n \times p$

Code C++:

```
#include <iostream>
#include <vector>
#include <iomanip>

using namespace std;

int** createMatrix (int row, int col)
{
    int **matrix = new int *[row];
    for (int i = 0; i < row; i++) {
        matrix[i] = new int [col];
    }
    return matrix;
}

void Nhap (int **matrix, int row, int col)
{
    cout << "Nhập số liệu (" << row << " dòng, mỗi dòng " << col << " phần
tu)" << endl;
    for (int i = 0; i < row; i++) {
        cout << "\tDòng " << i << ": ";
        for (int j = 0; j < col; j++) {
            cin >> matrix[i][j];
        }
    }
    cout << endl;
}
```

```

void print(int **a, int row, int col)
{
    if (a == nullptr) {
        cout << "Invalid print" << endl;
        return;
    }
    cout << "Ma tran ket qua: " << endl;
    for (int i = 0; i < row; i++) {
        cout << "\t\t|";
        for (int j = 0; j < col; j++) {
            cout << setw(3) << *(*(a + i) + j) << " ";
        }
        cout << setw(1) << "|" << endl;
    }
}

void deleteMatrix (int **a, int col)
{
    for (int i = 0; i < col; i++) {
        delete[] a;
    }
    delete[] a;
}

int** multipleMatrices (int **matrix1, int row1, int col1, int **matrix2, int
row2, int col2)
{
    if (col1 != row2) {
        cout << "Invalid" << endl;
        return nullptr;
    }
    int **matrixAns = createMatrix (row1, col2);
    for (int i = 0; i < row1; i++) {
        for (int j = 0; j < col2; j++) {
            int sum = 0;
            for (int k = 0; k < col1; k++) {
                sum += (*(*(matrix1 + i) + k) * *(*(matrix2 + k) + j));
            }
            *(*(matrixAns + i) + j) = sum;
        }
    }
    return matrixAns;
}

int main()
{
    int row1 = 4, col1 = 3;
    int row2 = 3, col2 = 2;
}

```

```

int **matrix1 = createMatrix(row1, col1); Nhap(matrix1, row1, col1);
int **matrix2 = createMatrix(row2, col2); Nhap(matrix2, row2, col2);

int **matrixAns = multipleMatrices(matrix1, row1, col1, matrix2, row2,
col2);
print(matrixAns, row1, col2);

deleteMatrix(matrix1, col1);
deleteMatrix(matrix2, col2);
deleteMatrix(matrixAns, col1);
return 0;
}

```

Lưu ý: Có thể làm giảm độ phức tạp của bài toán còn $O(n^C)$ sao cho $2 < C < 3$ bằng phương pháp chia để trị (Giải thuật Strassen), tuy ngoài mặt sự thay đổi này là không đáng kể nhưng với n đủ lớn thì đó lại chính là sự khác biệt vô cùng lớn.

Ví dụ 3:

Tính tích lớn nhất trong các tập con của mảng có N phần tử.

Cụ thể:

- Cho mảng có N phần tử, có 2^N tập con;
- Tính tích lớn nhất trong số 2^N tập con đó.

Lời giải:

Ý tưởng:

- Dùng giải thuật ngây thơ, tiếp cận bằng phương pháp Top – Down, nghĩa là cần xác định các hàm lớn trước khi xây dựng các hàm con nhỏ để hỗ trợ hàm lớn đó, kết quả của bước này là thu được một khung (sườn) chung cho cả bài toán;
- Ý tưởng ban đầu như sau:

- + Có một hàm chính tính tích, trả về kiểu int, thông số hàm là một vector đại diện cho mảng ban đầu (vector này được hình thành từ việc sao chép các phần tử từ mảng sang): *int maxMultiplication (vector <int> vct);*
- + Đề ý các trường hợp sau:
 - Nếu các phần tử đều dương: thì chỉ việc duyệt các phần tử + nhân tất cả lại \rightarrow tích;
 - Nếu các phần tử không âm (có ít nhất 1 số 0): thì duyệt các phần tử + nhân các phần tử khác 0 lại \rightarrow tích;
 - Nếu các phần tử âm/dương đầy đủ (kể cả số 0):
 - \triangleright Nếu số phần tử âm chẵn: thì duyệt các phần tử + nhân các phần tử khác 0 lại \rightarrow tích;
 - \triangleright Nếu số phần tử âm lẻ: thì duyệt các phần tử + nhân các phần tử khác 0 và nhân các phần tử âm lớn nhất (phần tử âm mà khi nhân lại làm tích không lớn nhất) \rightarrow tích;
- + *Như vậy, chia thành 2 trường hợp chính:*
 - *Số phần tử âm chẵn:*
 - \triangleright Nhân các phần tử khác 0
 - *Số phần tử âm lẻ:*
 - \triangleright Nhân các phần tử khác 0 (ngoại trừ phần tử âm lớn nhất):
 - Nếu chỉ có 1 phần tử âm lớn nhất;
 - Nếu có 2 phần tử âm lớn nhất trở lên;

Độ phức tạp: O(N).

Code C++:

```
#include <iostream>
#include <vector>

using namespace std;

// Hàm xử lí các phần tử 0 (nếu có thì xóa, đếm số phần tử 0)
int processing0Element(vector<int> &vct)
{
    int count = 0;
```

```

for (int i = vct.size() - 1; i >= 0; i--)
{
    if (vct[i] == 0)
    {
        count++;
        vct.erase(vct.begin() + i);
    }
}
return count;
}

// Hàm trả về 1 vector có 2 phần tử:
// Phần tử đầu tiên: số lượng số phần tử âm
// Phần tử tiếp theo: phần tử âm lớn nhất
vector<int> hasNegativeElement(vector<int> vct)
{
    vector<int> tmp;
    int count = 0, max = -10000, count1 = 0;
    for (int i : vct)
    {
        if (i < 0)
        {
            count++;
            if (i >= max)
            {
                if (i == max)
                {
                    count1++; // Đếm số phần tử lớn nhất bị lặp lại
                }           // Có 2 phần tử -2 lặp lại thì count1 = 1
                max = i;      // Tức có 1 phần tử -2 bị lặp lại với max
            }
        }
    }
    tmp.push_back(count); // Số lượng phần tử âm
    tmp.push_back(max);   // Phần tử âm lớn nhất
    tmp.push_back(count1); // Số lượng phần tử âm bị lớn nhất bị lặp lại
    return tmp;
}

// Hàm tính tích lớn nhất trong các tập con
int maxMultiplication(vector<int> vct, int N)
{
    int res = 1;
    int count_0 = processing0Element(vct);
    vector<int> processing = hasNegativeElement(vct);

    // Xét trường hợp biên (tất cả đều là số 0 hoặc 1 phần tử âm, còn lại 0)
    if (count_0 == N || (count_0 == N - 1 && processing[0] == 1))

```

```

    {
        return 0;
    }
    // Nếu số phần tử âm = 0 hoặc là số chẵn
    // res * (tất cả phần tử của vector)
    else if (processing[0] == 0 || processing[0] % 2 == 0)
    {
        for (int i : vct)
        {
            res *= i;
        }
    }
    // Nếu số phần tử âm là số lẻ
    // res * (tất cả phần tử của vector ngoại trừ phần tử âm lớn nhất)
    else
    {
        if (processing[2] == 0)
        { // Nếu chỉ có 1 số phần tử lớn nhất (0 bị lặp lại)
            for (int i : vct)
            {
                if (i != processing[1])
                {
                    res *= i;
                }
            }
        }
        else
        { // Nếu có 2 phần tử lớn nhất trở lên (bị lặp lại ít nhất 1 lần)
            for (int i : vct)
            {
                if (i != processing[1])
                {
                    res *= i;
                }
            }
            for (int i = 0; i < processing[2]; i++)
            {
                res *= processing[1];
            }
        }
    }
    return res;
}

int main()
{
    const int N = 7;
    int arr[N] = {0, 0, 3, 1, -4, -1, 8};

```

```

    vector<int> vct(arr, arr + N); // Sao chép mảng vào vector để xử lí
    cout << maxMultiplication(vct, N);
}

```

Cách tiếp cận tối ưu và hiệu quả hơn, giải quyết được rối rắm trong việc xét số phần tử âm lớn nhất là bao nhiêu để nhân theo từng trường hợp nhỏ, dễ sai, khó nâng cấp. Thuật toán như sau:

```

#include <iostream>
#include <vector>

using namespace std;

int maxMultiplication (int a[], int N)
{
    int count_0 = 0, count_Neg = 0;
    int max_Neg = -100000;
    int res = 1;
    for (int i = 0; i < N; i++) {
        if (a[i] == 0) {
            count_0++;
        }
        else {
            res *= a[i];
            if (a[i] < 0) {
                count_Neg++;
                max_Neg = max(max_Neg, a[i]);
            }
        }
    }

    if (count_0 == N || (count_Neg == 1 && count_0 == N - 1)) {
        return 0;
    }
    else if (count_Neg % 2 != 0) {
        res /= max_Neg;
    }
    return res;
}

int main()
{
    const int N = 9;
    int arr[N] = {2, 1, -5, 4, -3, -3, -3, -3, 0};
    cout << maxMultiplication(arr, N);
}

```

III. Giải thuật chia để trị (devide and conquer):

1. Tổng quan:

Chia để trị nói đúng hơn chỉ là 1 phương pháp lập trình chứ chưa đúng với nghĩa thuật toán.

Chia bài toán thành các bài toán con, mỗi bài toán là một phần của bài toán ban đầu, sau đó thực hiện một số công việc bổ sung để tính ra câu trả lời cuối cùng.

2. Các dạng bài toán dùng phương pháp chia để trị:

3. Phương pháp chung:

4. Các ví dụ cơ bản:

Ví dụ 1:

Bài toán MergeSort

Lời giải:

Để làm được bài toán này, cần tham khảo bài toán nhỏ hơn: [**Bài toán merge \(trộn\) 2 mảng 1 chiều với các phần tử có giá trị không giảm.**](#)

Bài toán có thể dùng giải thuật vét cạn để

Tư tưởng: Thuật toán sắp xếp trộn sử dụng phương pháp chia để trị, chia dãy ban đầu thành các dãy con cho tới khi dãy chỉ còn 1 phần tử, sau đó thực hiện trộn 2 dãy con tăng dần thành 1 dãy tăng dần với độ phức tạp tuyến tính.

Độ phức tạp: $O(n \log n)$ và tệ nhất cũng chỉ như thế. Khác với QuickSort thì tệ nhất sẽ có độ phức tạp $O(n^2)$.

Code C++:

MergeSort có cấu trúc như sau:

```
void mergeSort (int a[], int left, int right) {  
    if (left >= right) return;  
    int mid = left + (right - left) / 2;           //mid = (right + left)/2  
    mergeSort (a, left, mid);  
    mergeSort(a, mid + 1, right);  
    merge(a, left, mid, right);  
}
```

```
#include <iostream>  
#include <vector>  
  
using namespace std;  
  
void merge (int a[], int left, int mid, int right)  
{  
    vector <int> x(a + left, a + mid + 1);      //Xem giải thích bên dưới  
    vector <int> y(a + mid + 1, a + right + 1); //Xem giải thích bên dưới  
    int i = 0, j = 0;  
    while (i < x.size() && j < y.size())
```

```

{
    if (x[i] <= y[j]) {
        a[left++] = x[i++];
    }
    else {
        a[left++] = y[j++];
    }
}
while (i < x.size())
{
    a[left++] = x[i++];
}
while (j < y.size())
{
    a[left++] = y[j++];
}
}

void mergeSort (int a[], int left, int right)
{
    if (left >= right) {
        return;
    }
    int mid = left + (right - left)/2;
    mergeSort(a, left, mid);
    mergeSort(a, mid + 1, right);
    merge(a, left, mid, right);
}

int main()
{
    const int N = 10;
    int arr[N] {32, 14, 54, 6, 12, 1, 5, 6, 64, 41};
    mergeSort(arr, 0, N - 1);

    for (int i = 0; i < N; i++)
    {
        cout << *(arr + i) << " ";
    }
    return 0;
}

```

Giải thích đoạn code:

```

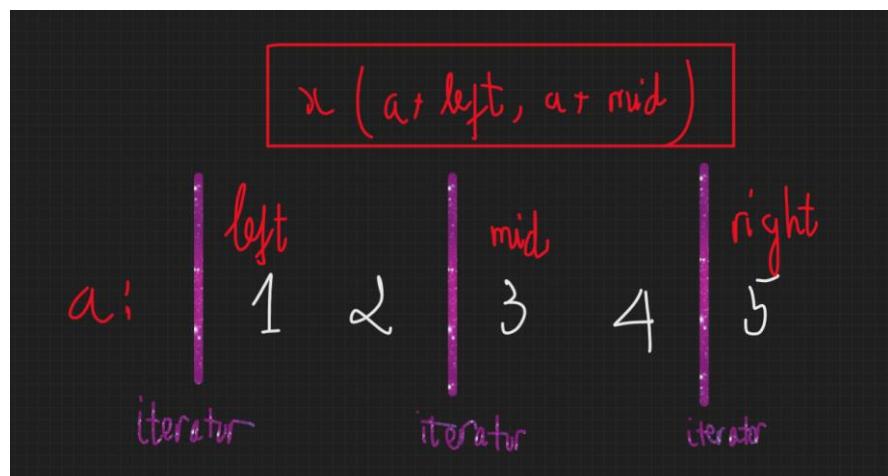
vector <int> x(a + left, a + mid + 1);      //Xem giải thích bên dưới
vector <int> y(a + mid + 1, a + right + 1); //Xem giải thích bên dưới

```

- Với a là một mảng đã cho thì cú pháp ở dòng code đầu tiên có nghĩa là tạo ra một vector x và copy các giá trị của các phần tử từ mảng a vào vector x, tương tự cho dòng code 2;
- 2 dòng code dùng để tạo “2 mảng con” trái và phải để thực hiện việc merge chúng lại;
 - + Vector x đại diện mảng con 1: copy giá trị từ vị trí left → mid;
 - + Vector y đại diện mảng con 2: copy giá trị từ vị trí mid + 1 → right;
- Tuy nhiên không thể dùng như sau:

```
vector <int> x(a + left, a + mid); //Bỏ + 1
vector <int> y(a + mid + 1, a + right); //BỎ + 1
```

vì với dòng 1: cú pháp đó có nghĩa tạo ra một iterator_1 (1 con trỏ) trỏ đến ngay trước vị trí left (ngay sau vị trí left – 1) và một iterator_2 trỏ đến ngay trước vị trí mid (ngay sau vị trí mid – 1) → chỉ copy từ left → mid – 1 thay vì left → mid → cộng 1 để thực hiện đúng thao tác. Cụ thể:



- Tương tự cách giải thích cho dòng 2.

Ví dụ 2: (Ứng dụng của ví dụ 1 – thuật toán mergeSort)

Đếm số cặp nghịch序 trong một mảng đã cho.

Lời giải:

Ý tưởng: Áp dụng phương pháp chia để trị thay vì vét cạn với 2 vòng lặp for để giảm độ phức tạp thời gian từ $O(n^2)$ còn $O(n \log n)$. Ta chỉ dùng thêm một biến đếm **count** vào hàm merge và một biến đếm **dem** vào hàm mergeSort vào code của ví dụ 1 để đếm số cặp nghịch序 sinh ra khi gộp.

Code C++:

```
#include <iostream>
#include <vector>

using namespace std;

int merge (int *a, int left, int mid, int right)
{
    vector <int> x (a + left, a + mid + 1);
    vector <int> y (a + mid + 1, a + right + 1);
    int i = 0, j = 0;
    int count = 0;

    while (i < x.size() && j < y.size())
    {
        if (x[i] <= y[j]) {
            *(a + left++) = x[i++];
        }
        else {
            count += x.size() - i;
            *(a + left++) = y[j++];
        }
    }
    while (i < x.size())
    {
        *(a + left++) = x[i++];
    }
    while (j < y.size())
    {
        *(a + left++) = y[j++];
    }
    return count;
}

int mergeSort (int *a, int left, int right)
{
    int dem = 0;
    if (left >= right) return dem; //Mang 1 phan tu thi so CNT = 0
    int mid = left + (right - left)/2;
    dem += mergeSort(a, left, mid); //Cap NT mang ben trai
    dem += mergeSort(a, mid + 1, right); //Cap NT mang ben phai
    dem += merge(a, left, mid, right); //Cap NT gom 1 phan tu ben trai, 1 ptu
ben phai
    return dem;
}
```

```

int main()
{
    const int N = 4;
    int *arr = new int [N] {5, 6, 2, 3};
    cout << mergeSort(arr, 0, 3);

    delete[] arr;
    return 0;
}

```

Ví dụ 3:

Bài toán QuickSort

Lời giải:

Ý tưởng: Áp dụng phương pháp chia để trị (devide and conquer) tương tự như bài toán mergeSort ở ví dụ 1, chia mảng đã cho làm 2 mảng con (bằng phân hoạch Lomuto hoặc bằng phân hoạch Hoare). Sau đó thực hiện hàm quickSort phù hợp với phân hoạch đã dùng.

- Phân hoạch Lomuto đơn giản, dễ cài đặt tuy nhiên không tốt bằng phân hoạch Hoare;
- Phân hoạch Hoare chính là phương pháp chia mảng được tác giả viết chính thức.

Độ phức tạp: - $O(n \log n)$ (trung bình)

- $O(n^2)$ (tê nhất)

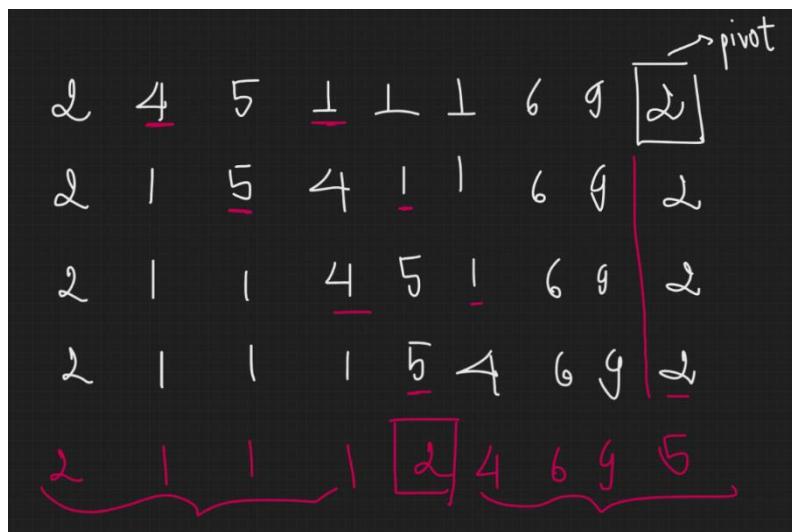
1. *Phân hoạch Lomuto + giải thuật:*

- Phương pháp:
 - + Chọn pivot là phần tử cuối của mảng;
 - + Chia mảng thành 2 phần sao cho phần bên trái gồm các phần tử nhỏ hơn hoặc bằng pivot, phần bên phải gồm các phần tử lớn hơn pivot.
 - + Xét mảng con trái, thực hiện lại (+ đầu tiên) cho đến khi mảng con có 1 phần tử;

- + Xét mảng con phải, thực hiện lại (+đầu tiên) cho đến khi mảng con có 1 phần tử;
- + Kết thúc. (không cần bước combine như mergeSort vì mảng vốn đã được sắp xếp trực tiếp thông qua các bước trên).

Mô tả giải thuật:

- Cho i, j là chỉ số ảo duyệt từ đầu mảng đến cuối mảng.
- Tìm 1 cặp nghịch thế (x_i, y_j) đầu tiên trong mảng đã cho ($i < j$, tức x nằm trước y trong mảng) sao cho $x_i > \text{pivot}$ và $y_j < \text{pivot}$.
- Cặp nghịch thế này đã vi phạm quy tắc sắp xếp nên ta swap x_i và y_j để chúng về đúng vị trí, số nhỏ sau khi swap (y_j) sẽ đứng trước pivot, số lớn sau khi swap (x_i) sẽ đứng sau pivot.
- Tiếp tục cho đến khi không còn cặp nghịch thế nào thỏa mãn.



- Kết quả thu được là một mảng bị chia đôi bởi pivot (không nhất thiết phải bằng nhau như mergeSort), phần trước pivot bao gồm những phần tử nhỏ hơn hoặc bằng pivot, phần sau pivot bao gồm những phần tử lớn hơn pivot.

Pseudo – code:

```

int partition (int a[], int left, int right){}
void quickSort (int a[], int left, int right) {
    if (left >= right) return;
    int p = partition(a, left, right);      //p: chỉ số (pivot) sau khi chia
    quickSort (a, left, p - 1);
}

```

```

    quickSort(a, p + 1, right);
}

```

Code C++:

```

int partition (int *a, int left, int right)
{
    int pivot = *(a + right);
    int i = left - 1;
    for (int j = left; j < right; j++) {
        if (*(a + j) <= pivot) {
            i++;
            swap(*(a + i), *(a + j));
        }
    }
    i++;
    swap(*(a + i), *(a + right));
    return i;
}

void quickSort (int *a, int left, int right)
{
    if (left >= right) return;
    int p = partition(a, left, right);
    quickSort(a, left, p - 1);
    quickSort(a, p + 1, right);
}

```

Lưu ý: Trong hàm partition lệnh swap trước return bắt buộc phải là swap (*(a + i), *(a + right)), không được đổi thành swap (*(a + i), pivot). Lý do: pivot chỉ là biến tạm sao chép kết quả của *(a + right) chứ không phải *(a + right), nên swap (*(a + i), pivot) chỉ đổi *(a + i) với giá trị của pivot mà KHÔNG LÀM THAY ĐỔI giá trị của *(a + right). Tuy nhiên, nếu muốn swap với pivot thì phải khai báo pivot là tham khảo với dấu &. Cụ thể:

```
int &pivot = *(a + right);
```

Khi đó kết quả thu được là không đổi.

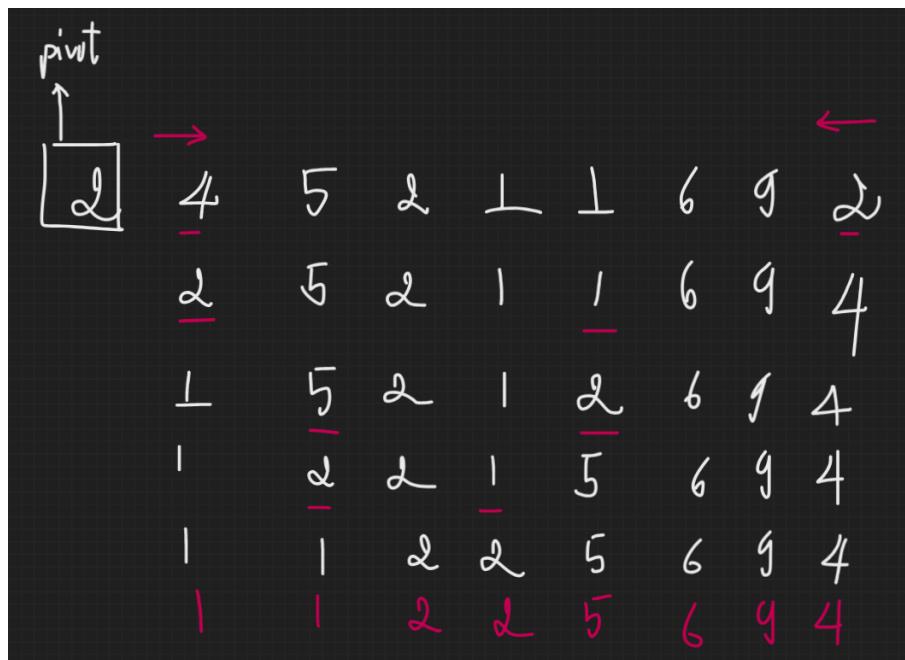
2. Phân hoạch Hoare + giải thuật:

- Phương pháp:
 - + Dùng 2 vòng lặp do – while thay đổi biến i, j (cụ thể xem code dưới)
 - + Chọn pivot = a[left]

- + Ý tưởng là tìm cặp nghịch thế (x, y) thỏa mãn $x \geq \text{pivot}$ và $y \leq \text{pivot}$, sau đó swap x và y để có được thứ tự: $y < \text{pivot} < x$
- + Vòng lặp i, j được thực hiện liên tục cho đến khi không còn cặp nghịch thế nào thỏa mãn, tức $i \geq j$ sẽ dừng.

Mô tả giải thuật:

- Cho i, j là các chỉ số ảo, i duyệt từ đầu \rightarrow cuối mảng, j duyệt từ cuối \rightarrow đầu mảng.
- Tìm 1 cặp nghịch thế (x_i, y_j) đầu tiên trong mảng đã cho ($i < j$, tức x nằm trước y trong mảng) sao cho $x_i > \text{pivot}$ và $y_j < \text{pivot}$.
- Cặp nghịch thế này đã vi phạm quy tắc sắp xếp nên ta swap x_i và y_j để chúng về đúng vị trí, số nhỏ sau khi swap (y_j) sẽ đứng trước pivot, số lớn sau khi swap (x_i) sẽ đứng sau pivot.
- Tiếp tục cho đến khi không còn cặp nghịch thế nào thỏa mãn.



- Kết quả thu được là một mảng vị chia đôi KHÔNG NHẤT THIẾT bởi pivot (và cũng không nhất thiều phải bằng nhau như mergeSort), phần trước pivot bao gồm những phần tử nhỏ hơn hoặc bằng pivot, phần sau pivot bao gồm những phần tử lớn hơn hoặc bằng pivot. (lưu ý: các phần tử của phần trước & phần sau có thể bằng pivot).

Pseudo – code:

```
int partition (int a[], int left, int right) {}

void quickSort (int a[], int left, int right) {
    if (left >= right) return;
    int p = partition (a, left, right);
    quickSort (a, left, p);           //Với Lomuto thì p – 1 thay vì p
    quickSort(a, p + 1, right);
}
```

Code C++:

```
int partition (int *a, int left, int right)
{
    int pivot = *(a + left);
    int i = left - 1, j = right + 1;
    while (true)
    {
        do {
            i++;
        }
        while (*(a + i) < pivot);
        do {
            j--;
        }
        while (*(a + j) > pivot);
        if (i < j)
        {
            swap(*(a + i), *(a + j));
        }
        else return j;
    }
}

void quickSort (int *a, int left, int right)
{
    if (left >= right) return;
    int p = partition(a, left, right);
    quickSort(a, left, p);
    quickSort(a, p + 1, right);
}
```

Lí do sử dụng do – while thay vì while thông thường: Trong thuật toán này bắt buộc dùng do – while để lặp vì thuật toán sẽ sai (hoặc lặp vô tận) với trường hợp như sau:

4 8 4 | | | 3 6 2

Sau 1 lúc thực hiện (điều kiện) trước (lệnh) sau thì sẽ có $a[i] = 4$, $a[j] = 4$ và $\text{pivot} = 4$, khi đó điều kiện ($a[i] < \text{pivot}$) không được thực hiện \rightarrow sang vòng lặp while của $j \rightarrow$ cũng không thực hiện được do không thỏa ($a[j] > \text{pivot}$) $\rightarrow i < j$ nên vòng lặp while (true) được thực hiện tiếp \rightarrow điều kiện ($a[i] < \text{pivot}$) không thực hiện $\rightarrow \dots \rightarrow$ lặp vô tận.

(4)
2 3 4 4 8 6 8
i i i j j j j

Nếu muốn dùng while() thì nên sửa điều kiện ($< \rightarrow <=$) và ($> \rightarrow >=$).

Ví dụ 3:

Bài toán *Tìm kiếm Nhị phân (Binary Search)* cho mảng đã được sắp xếp tuyến tính

Lời giải:

Ý tưởng: **(Nguồn chatGPT-4)** Thuật toán tìm kiếm nhị phân có thể được triển khai dưới hai phiên bản chính: đệ quy và lặp.

- **Tìm kiếm nhị phân đệ quy:** Trong phiên bản này, thuật toán được triển khai thông qua hàm đệ quy. Thuật toán chia mảng thành hai nửa, so sánh giá trị ở giữa mảng với giá trị cần tìm. Dựa vào kết quả so sánh, thuật toán tiếp tục gọi đệ quy trên nửa mảng phù hợp để tiếp tục tìm kiếm. Đệ quy tiếp tục cho đến khi

tìm thấy giá trị hoặc mảng đã được thu hẹp đến kích thước không thể thu hẹp được nữa.

- Tìm kiếm nhị phân lặp: Trong phiên bản này, thuật toán được triển khai thông qua vòng lặp. Thuật toán chia mảng thành hai nửa, so sánh giá trị ở giữa mảng với giá trị cần tìm. Dựa vào kết quả so sánh, thuật toán tiếp tục chỉnh sửa ranh giới của mảng để thu hẹp phạm vi tìm kiếm. Vòng lặp tiếp tục cho đến khi tìm thấy giá trị hoặc mảng đã được thu hẹp đến kích thước không thể thu hẹp được nữa.

Độ phức tạp: Cả hai phiên bản tìm kiếm nhị phân đều có độ phức tạp là $O(\log N)$, với N là kích thước của mảng. Sự lựa chọn giữa đệ quy và lặp thường phụ thuộc vào sự ưu tiên của ngôn ngữ lập trình và yêu cầu cụ thể của vấn đề được giải quyết.

Code C++:

Tìm kiếm chỉ số của phần tử bằng tìm kiếm nhị phân đệ quy:

```
int binarySearch_recursion (int *a, int left, int right, int value)
{
    if (left > right) return -1;
    int mid = (left + right)/2;
    if (* (a + mid) > value) {
        return binarySearch_recursion(a, left, mid - 1, value);
    }
    else if (* (a + mid) < value) {
        return binarySearch_recursion(a, mid + 1, right, value);
    }
    else if (* (a + mid) == value) {
        return mid;
    }
}
```

Lưu ý: khi xét điều kiện thỏa mãn ($left > right$), ta không được thêm dấu bằng vào điều kiện, vì khi đó có thể dẫn đến kết quả sai.

Ví dụ: Giả sử ta xét điều kiện ($left \geq right$) thì return -1. Khi mảng chỉ đang xét đến $[4, 5, 8]$, $left$ tương ứng với 4, mid tương ứng với 5, $right$ tương ứng với 8 và số cần tìm là 8. Ta gọi hàm tương ứng với trường hợp $*(arr + mid) < value$, tức return hàm `binarySearch_recursion(a, mid + 1, right, value)`. Trong lời gọi hàm đó, $left$ và

right bằng nhau, ngay lập tức -1 được trả về, thay vì tiếp tục chương trình để chỉ ra chỉ số của 8 trong mảng.

Tìm kiếm chỉ số của phần tử bằng tìm kiếm nhị phân vòng lặp:

```
int binarySearch_loop (int *arr, int N, int value)
{
    int left = 0;
    int right = N - 1;
    int mid;
    while (left <= right) {
        mid = left + (right - left)/2;
        if (*(arr + mid) > value) {
            right = mid - 1;
        }
        else if (*(arr + mid) < value) {
            left = mid + 1;
        }
        else return mid;
    }
    return -1;
}
```

Ưu điểm: Độ phức tạp là O(logn) thay vì O(n) nếu như dùng vòng lặp duyệt qua từng phần tử và kiểm tra điều kiện bằng.

Lưu ý: trong điều kiện vòng lặp while, toán tử cần có chính là \leq , không phải $<$. Nếu sử dụng $<$, điều đó có thể gây ra lỗi. Tương tự như lỗi có thể mắc phải như trên. Ta có thể hiểu khi thực hiện bài toán tìm kiếm nhị phân, điều kiện luôn là khi nào $left > right$ thì mới dừng, trường hợp $left = right$ vẫn phải tiếp tục thực hiện bài toán.

Ví dụ 4:

Lũy thừa nhị phân

Tìm lũy thừa bậc N của x (tính x^N , với x, N cho trước).

Lời giải:

Ý tưởng: Bài toán có thể dùng giải thuật Ngây thơ (naïve) xét tất cả các khả năng, rồi cho ra kết quả cuối cùng với vòng lặp N lần tương ứng nhân x với N lần, khi đó độ phức tạp thời gian là tuyến tính – O(n), cụ thể:

```
int square (int x, int N) {
    int res = 1;
    for (int i = 0; i < N; i++) {
```

```

        res *= x;
    }
    return res;
}

```

Tuy nhiên, bài toán có thể giải quyết với độ phức tạp thời gian ít hơn, hiệu quả hơn với phương pháp chia để trị. Với tư tưởng đơn giản là:

$$x^N = \begin{cases} x^{N-1} \cdot x & (N - le) \\ x^{\frac{N}{2}} \times x^{\frac{N}{2}} & (N_chan) \end{cases}$$

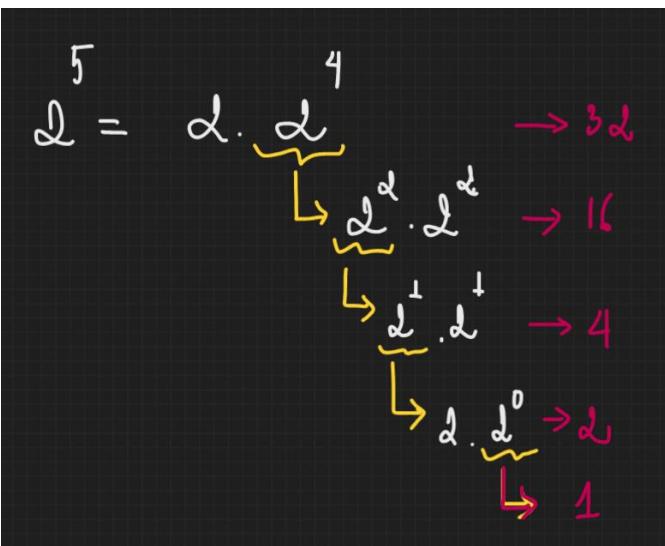
Lũy thừa nhanh (lũy thừa nhị phân) bằng đệ quy:

```

double fastPower (double x, unsigned short N)
{
    if (N == 0) return 1;
    if (N % 2 == 1 ) return x * fastPower(x, N - 1);
    double y = fastPower(x, N/2);
    return y*y;
}

```

Mô tả quá trình hoạt động:



Lũy thừa nhanh (lũy thừa nhị phân) bằng vòng lặp while:

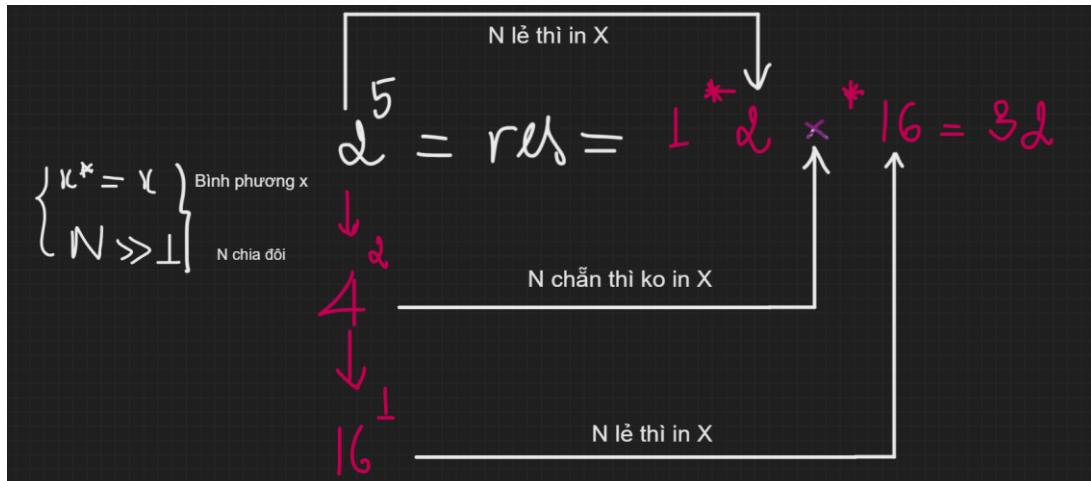
```

double fastPower (double x, unsigned short N)
{
    double res = 1;
    while (N) {
        if (N % 2 == 1) res *= x;
        x *= x;
        N /= 2;
    }
    return res;
}

```

}

Mô tả quá trình hoạt động:



Độ phức tạp: O(log n)

IV. Giải thuật tham lam (greedy):

1. *Tổng quan:*

Giải thuật tham lam là bất kỳ thuật toán nào thực hiện dựa trên chiến lược lựa chọn giải pháp tối ưu cục bộ ở mỗi bước:

- Lựa chọn tối ưu ở đây phụ thuộc vào yêu cầu của bài toán;
- Cục bộ nghĩa là lựa chọn tức thì tại bất kỳ lần lặp nào.

Hầu hết các bài toán về giải thuật tham lam sẽ yêu cầu giá trị tối đa hoặc tối thiểu của một thứ gì đó.

Giải thuật tham lam (greedy algorithm) là một phương pháp giải quyết bài toán bằng cách luôn chọn lựa giải pháp tốt nhất tại mỗi bước, hy vọng rằng việc lựa chọn tốt nhất ở mỗi bước sẽ dẫn đến giải pháp tốt nhất cho toàn bộ bài toán.

Cách tiếp cận tham lam thường được sử dụng trong các bài toán tối ưu hóa, nơi mục tiêu là tìm một giải pháp tốt nhất dựa trên một tiêu chí định trước. Thuật toán tham lam đưa ra quyết định dựa trên thông tin có sẵn tại thời điểm hiện tại mà không quan tâm đến tác động của quyết định đó tới các bước tiếp theo.

Rất nhiều giải thuật nổi tiếng được thiết kế dựa trên ý tưởng tham lam, ví dụ như giải thuật cây khung nhỏ nhất của Dijkstra, giải thuật cây khung nhỏ nhất của Kruskal, ...

2. *Tính chất chung:*

- Mỗi lựa chọn tối ưu cục bộ có thể phụ thuộc vào lựa chọn trước đó;
- Mỗi lựa chọn tối ưu cục bộ không phụ thuộc vào lựa chọn sau đó (lựa chọn trong tương lai) và không phụ thuộc vào các lựa chọn tương đương;
- Đa số các vấn đề đều có mục tiêu tìm ra cái tối đa, tối thiểu của một thứ gì đó, và do đó dữ liệu đầu vào cần được sắp xếp trước khi lựa chọn (trong đa số dạng bài có thể sử dụng tham lam);
- “Lựa chọn” được chọn bởi “quyết định”;
- Thuật toán không quay lại xét các quyết định trước đó (quyết định trong quá khứ) như Quy hoạch động (Dynamic programming);
- Việc quyết định sớm và thay đổi hướng đi của giải thuật cùng với việc không bao giờ xét lại các quyết định cũ sẽ dẫn đến kết quả là giải thuật này không tối ưu để tìm giải pháp toàn cục.

3. *Phương pháp giải tổng quát:*

Tham lam là một thuật toán mơ hồ, tuy nhiên nếu nhận diện được một bài toán có thể giải quyết bằng tham lam, thì đây là cách tiếp cận nên xét:

- Xác định đâu là các lựa chọn (thường xác định: Để “tối thiểu” yêu cầu, cần “tối đa” điều gì, hoặc ngược lại, ...);
- Sắp xếp các lựa chọn (từ tốt nhất → tệ nhất);
- Duyệt qua các lựa chọn đã được sắp xếp, thực hành giải quyết vấn đề ở bước đó;
- Kết thúc.

Đó là cách tiếp cận (sắp xếp → duyệt → kết thúc) dùng cho những bài các lựa chọn có tính chất tốt – tệ không đổi xuyên suốt quá trình, tuy nhiên khi các tính chất đó là không xuyên suốt (sự tốt – tệ thay đổi sau mỗi quyết định 1 lựa chọn tối ưu), thì quy trình cần sử lại như sau: (sắp xếp → duyệt 1 lựa chọn tối ưu →

sắp xếp → duyệt 1 lựa chọn tối ưu (trong các lựa chọn còn lại) → sắp xếp → ...).

4. Các ví dụ điển hình:

Ví dụ 1:

Bài toán đổi tiền:

Cho các tờ tiền mệnh giá lần lượt: 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000.

Số lượng mỗi loại tờ là vô số.

Giả sử cần đổi số tiền Value (ví dụ 43.200) bằng các loại tờ tiền trên, hỏi sau khi đổi thì số tờ tiền tối thiểu là bao nhiêu? Tức tìm số các loại tờ tiền ít nhất đủ để đổi số tiền Value đã cho.

Lời giải:

Ý tưởng: Đây là bài toán kinh điển của thuật toán tham lam, khi tìm ra lựa chọn giải pháp tối ưu cục bộ (lựa chọn tốt nhất ở tại thời điểm lựa chọn và đổi hướng giải quyết bài toán bằng quyết định 1 lựa chọn cụ thể):

- Lựa chọn ở đây là lựa chọn số loại tờ tiền mỗi loại, lần lượt cho đến khi đủ số tiền Value;
- Để “tối thiểu” số loại tờ tiền cần sử dụng, cần “tối đa” mệnh giá của mỗi loại tờ tiền được chọn (ví dụ với số tiền Value = 43100 thì cần 43 tờ tiền mệnh giá 1000 và 1 tờ tiền mệnh giá 100, thay vì 43100 tờ mệnh giá 1, đó chính là lựa chọn tối ưu);
- Giải pháp:
 - + Cho 1 vòng lặp for lặp với số lần m (m là số loại tờ tiền để cho, duyệt qua tất cả các lựa chọn), thứ tự sắp xếp các lựa chọn từ tốt nhất → tệ nhất là 1000, 500, 200, 100, 50, 20, 10, 5, 2, 1, lý do là vì thỏa tính chất “tối đa” mệnh giá ở mỗi lựa chọn loại tờ tiền. Như vậy lặp số lần m = 10.
 - + Ở mỗi lần lặp, ta cần xác định số lượng tờ tiền tối đa có thể sử dụng của loại đó, đó chính là dấu hiệu của toán chia lấy nguyên, như vậy ta lấy Value / (mệnh giá tiền loại đang xét). Khi đó phần

nguyên **q** chính là số lượng tờ tiền tối đa có thể sử dụng, phần dư **r** chính là lượng tiền còn lại, tức Value còn lại sau khi đã dùng **q** tờ tiền;

- + Tiếp tục vòng lặp với tờ tiền mệnh giá tiếp theo với Value hiện tại là **r**, cho đến khi Value = 0 (đã đủ số tiền để đổi).

Độ phức tạp: O(N) với N = m = số loại tờ tiền đã cho.

Code C++:

```
#include <iostream>

using namespace std;

class Solution {
public:
    int MoneyGreedy (int*, int, int);
};

int Solution :: MoneyGreedy (int *a, int N, int value)
{
    int count = 0; // Ket qua so to tien toi thieu;
    for (int i = 0; i < N; i++) {
        count += value / *(a + i);
        value %= *(a + i);

        if (value == 0) {
            break;
        }
    }
    return count;
}

int main()
{
    const int N = 10;
    int *arr = new int [N] {1000, 500, 200, 100, 50, 20, 10, 5, 2, 1};
    Solution solution;
    cout << solution.MoneyGreedy(arr, N, 103);
}
```

Nếu muốn in ra các loại tờ tiền đã sử dụng, thì sửa code như sau:

```
int Solution ::MoneyGreedy(int *a, int N, int value)
{
    int count = 0; // Ket qua so to tien toi thieu;
    for (int i = 0; i < N; i++)
```

```

{
    count += value / *(a + i); // Số lượng tờ tiền được chọn tối đa

    if (value / *(a + i))
    { // Nếu count thay đổi, tức mệnh giá này có sử dụng thì mới in
        for (int j = 0; j < value / *(a + i); j++) {
            cout << *(a + i) << " ";
        }
    }

    value %= *(a + i);

    if (value == 0) {
        break;
    }
}
cout << endl;
return count;
}

```

Hạn chế của thuật toán tham lam:

Tham lam chỉ đưa ra lựa chọn giải pháp tối ưu cục bộ, nên không đảm bảo sẽ đưa ra đáp án tối ưu toàn cục, tức lựa chọn ở thời điểm đó là tốt nhất, nhưng đến cùng lại không cho ra đáp án tốt nhất, trong bài toán trên ta thấy rõ ở ví dụ sau:

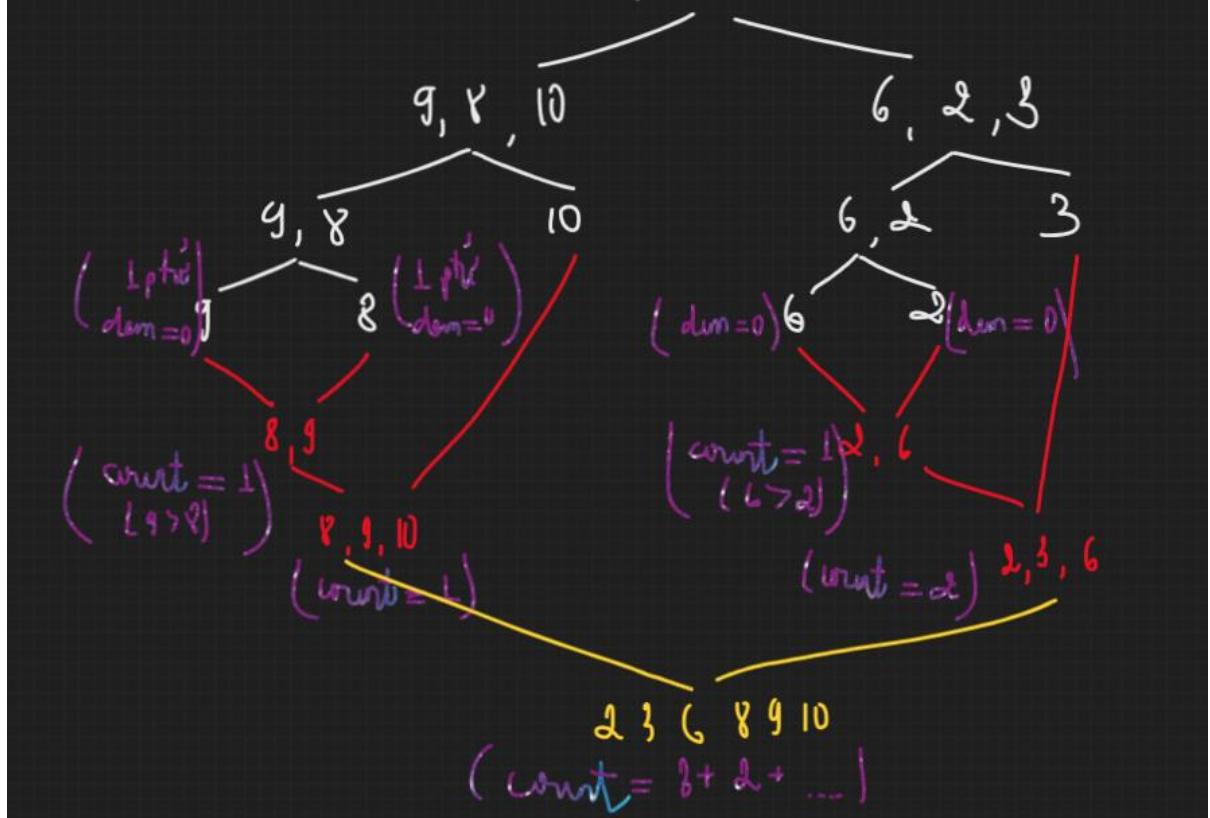
Khi ta có 4 loại tờ tiền mệnh giá 6, 5, 4, 1 và số tiền cần đổi Value = 9. Thì đáp án thuật toán đưa ra là 6, 1, 1, 1 thay vì 5, 4. Rõ ràng, đáp án 6, 1, 1, 1 là không tối ưu so với 5, 4, nên đáp án đã sai. Lý do là vì ở bước đầu tiên, lựa chọn tối ưu là 6, nên 6 được chọn là hợp lí, sau đó 5, 4 là lựa chọn không hợp lệ và cuối cùng là lựa chọn 1 với số lần được chọn là 3. Khi đến tờ mệnh giá 4, thuật toán đã không xét lại quá khứ để loại bỏ 6 và thay vào 5 để được kết quả tối ưu hơn, nói một cách tổng quát, giải thuật tham lam trong trường hợp này đã bỏ qua viễn cảnh tương lai khi không quay về quá khứ để suy xét lại kết quả dẫn đến việc kết cục cuối cùng không tối ưu như mong muốn.

Để giải quyết bài toán này, ta cần sử dụng đến thuật toán Quy hoạch động (Dynamic Programming).

V. Giải thuật quy hoạch động (dynamic programming – dp):

1. *Tổng quan:*

$$a[] = \{ 9, 8, 10, 6, 2, 3 \}$$



B. CÁC THUẬT TOÁN CHÍNH:

I. TÌM KIẾM VÀ SẮP XẾP (SEARCH & SORT)

1. Tìm kiếm:

1.1 Tìm kiếm tuyến tính (linear search):

Ý tưởng: Là một dạng toán sử dụng giải thuật vét cạn, duyệt qua từng trường phần tử trong mảng và xét điều kiện, nếu tồn tại trả về chỉ số tại phần tử đó, ngược lại trả về -1.

Độ phức tạp: $O(N)$, tốt nhất $O(1)$ (nếu giá trị cần tìm nằm ở đầu mảng) và xấu nhất $O(N)$ khi mảng không tồn tại giá trị cần tìm.

Code C++:

```
int linearSearch (int *arr, int N, int value)
{
    for (int i = 0; i < N; i++)
    {
        if (*(arr + i) == value) {
            return i;
        }
    }
    return -1;
}
```

1.2 Tìm kiếm nhị phân (binary search):

Độ phức tạp: $O(\log N)$, tốt nhất $O(1)$ (nếu giá trị cần tìm nằm vị trí mid của mảng) và xấu nhất $O(\log N)$ khi mảng không tồn tại giá trị cần tìm.

Ý tưởng & Code C++: Tham khảo tại [đây](#).

1.3 Tìm kiếm nội suy (Interpolation search):

Ý tưởng:

- Điểm khác biệt duy nhất giữa nội suy và nhị phân nằm ở việc tính mid. Với tìm kiếm nhị phân, mid luôn là vị trí giữa mảng đang xét, nhưng với tìm kiếm nội

suy thì có thể không như vậy, mid của nội suy sẽ là một vị trí tương đối gần đúng của giá trị cần tìm trong khoảng giới hạn.

- Tìm kiếm nhị phân sử dụng *left*, *right* để tính mid là vị trí giữa, tìm kiếm nội suy sử dụng *left*, *right*, *arr[left]*, *arr[right]* (thông tin về phân phôi dữ liệu) để tính mid là một vị trí xấp xỉ vị trí của giá trị cần tìm.
- Công thức tính mid được lấy từ công thức tìm kiếm tuyến tính nội suy:

$$mid = left + \frac{(right - left) \times (value - arr[left])}{arr[right] - arr[left]}$$

- Nội suy chỉ có lợi khi dữ liệu phân bố đồng đều, ngược lại tìm kiếm nhị phân cho biểu hiện tốt hơn.

Độ phức tạp: O(logN), tốt nhất O(1) khi giá trị cần tìm nằm tại mid, xấu nhất O(N) khi *arr[right] – arr[left]* có giá trị quá lớn (2 giá trị cách biệt lớn trong khi nằm gần vị trí nhau (mảng phân bố không đều) làm cho phân số trong công thức bằng 0, lúc đó mid sẽ tăng dần từng đơn vị,) → tìm tuyến tính → O(N).

Code C++: <lặp>

```
int interpolationSearch (int *arr, int N, int value)
{
    int left = 0;
    int right = N - 1;
    int mid;
    while (left <= right) {
        mid = left + ((double)(right - left) * (value - *(arr +
left)))/(*(arr + right) - *(arr + left));
        if (*(arr + mid) == value) {
            return mid;
        }
        else if (*(arr + mid) < value) {
            left = mid + 1;
        }
        else {
            mid = right - 1;
        }
    }
    return -1;
}
```

2. Sắp xếp:

2.1 Sắp xếp bằng phương pháp lựa chọn trực tiếp (Selection Sort):

Ý tưởng: Vết cạn bằng 2 vòng lặp for duyệt qua từng cặp phần tử, sắp xếp tăng/giảm dần theo yêu cầu:

Độ phức tạp: $O(N^2)$.

Code C++:

```
void selectionSort (vector<int> &a)
{
    int N = a.size();
    for (int i = 0; i < N; i++) {
        int min = i;
        for (int j = i + 1; j < N; j++) {
            min = (a[j] < a[min]) ? j : min;
        }
        swap(a[i], a[min]);
    }
}
```

2.2 Sắp xếp bằng phương pháp nổi bọt (Bubble Sort):

Ý tưởng: Sắp xếp số lớn nhất trước – mỗi vòng lặp đưa số lớn nhất trong mảng về cuối. Thực hiện N vòng lặp tương tự cho đến khi toàn bộ phần tử được sắp xếp.

Độ phức tạp: $O(N^2)$.

Code C++:

```
void bubbleSort (int *arr, const int N)
{
    for (int step = 0; step < N; step++) {
        for (int i = 1; i < N - step; i++) {
            if (*(arr + i) <= *(arr + i - 1)) {
                swap(*(arr + i), *(arr + i - 1));
            }
        }
    }
}
```

Lưu ý: Ở vòng lặp con, nên duyệt i từ 1 đến $N - step$, vì ngay tại bước thứ ‘step’, đã có ‘step’ phần tử được sắp xếp đúng vị trí, do vậy số phần tử cần sắp xếp hiện tại chỉ còn $N - step$.

Thuật toán Bubble Sort tối ưu: thuật toán sắp xếp nổi bọt nguyên thủy như trên sẽ không bỏ qua trường hợp duyệt mảng đã được sắp xếp xong tại một thời điểm nào đó trong quá trình chạy code. Tức số lần duyệt bị cố định trong suốt chương trình, để kết thúc thuật toán khi mảng đã được sắp xếp hoàn chỉnh tại step nào đó, ta cần bổ sung biến **swapped** kiểm tra xem mảng đã được sắp xếp hoàn chỉnh tại step đó hay chưa. Nhận thấy: tại bước thứ step, nếu mảng đã được sắp xếp hoàn chỉnh thì sẽ không có câu lệnh swap nào được thực thi, đồng nghĩa với việc không còn phần tử nào đứng sai vị trí để swap, tức mảng đã sắp xếp xong, đó chính là lúc kết thúc thuật toán.

Code C++:

```
void bubbleSort (int *arr, const int N)
{
    for (int step = 0; step < N; step++) {
        bool swapped = false;
        for (int i = 1; i < N - step; i++) {
            if (*(arr + i) <= *(arr + i - 1)) {
                swap(*(arr + i), *(arr + i - 1));
                swapped = true;
            }
        }
        if (swapped == false) break;
    }
}
```

2.3 Sắp xếp bằng phương pháp mắc lược (Comb Sort);

Ý tưởng:

- Đây là phiên bản cải tiến của thuật toán sắp xếp nổi bọt (Bubble Sort);
- Ưu điểm so với Bubble Sort:
 - + Đưa các phần tử có giá trị nhỏ về trước tránh trường hợp rất nhiều phần tử nhỏ nằm ở cuối mảng, với Bubble Sort sẽ phải so sánh và swap rất nhiều lần;
 - + ...
- Sự khác biệt so với Bubble Sort:
 - + Khoảng cách (gap) giữa các phần tử khi so sánh lớn hơn 1 (với Bubble Sort so sánh giữa các phần tử liền kề nhau);

+ ...

Độ phức tạp: O(N) tốt nhất. O(N^2) trung bình và xấu nhất.

Code C++:

```
/* #include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>

using namespace std;

class BubbleSort
{
    const int N = 50;
    vector<int> arr;
public:
    BubbleSort() : arr(N) {
        srand(time(NULL));
        for (int i = 0; i < N; i++) {
            arr[i] = rand() % 100;
        }
    }
    friend void print (BubbleSort&);
    friend void CombSort(BubbleSort&);
    void bubbleSort();
};

void BubbleSort::bubbleSort()
{
    for (int step = 0; step < N; step++) {
        bool swapped = false;
        for (int i = 1; i < N - step; i++) {
            if (arr[i - 1] > arr[i]) {
                swap(arr[i - 1], arr[i]);
                swapped = true;
            }
        }
        if (swapped == false) {
            break;
        }
    }
} */

void CombSort (BubbleSort &BBS)
{
    double shrinkFactor = 1.3;
```

```

int gap = BBS.N;

while (gap > 1)
{
    gap = max(1, static_cast<int>(gap/shrinkFactor));
    for (int i = 0; i + gap < BBS.N; i++) {
        if (BBS.arr[i] > BBS.arr[i + gap]) {
            swap(BBS.arr[i], BBS.arr[i + gap]);
        }
    }
}
/*
void print (BubbleSort &BBS)
{
    for (int i : BBS.arr) {
        cout << i << " ";
    }
    cout << endl;
}

int main()
{
    BubbleSort BBS;
    print(BBS);
    CombSort(BBS);
    print(BBS);
} */

```

2.4 Sắp xếp bằng phương pháp chèn trực tiếp (Insertion Sort):

Ý tưởng: Vẫn là 2 vòng lặp for duyệt qua các cặp phần tử trong mảng, tuy nhiên chèn trực tiếp phức tạp hơn về cách thực thi:

- Mỗi vòng lặp for mẹ ở vị trí i làm cho mảng được sắp xếp đúng theo thứ tự đến vị trí i , duyệt hết vòng lặp tức mảng được sắp xếp đúng theo thứ tự hoàn toàn;
- Với vòng lặp tại vị trí i , **lưu ý rằng khi đã lặp đến i thì tất cả phần tử từ 0 đến i đều đã được sắp xếp.** Lấy $a[i]$ làm mốc, xét các phần tử trước $a[i]$ bằng một vòng lặp con với chỉ số k , nếu gặp $a[k] < a[i]$ thì từ

vị trí 0 đến k đã xếp đúng vị trí, ta break; thoát vòng lặp k, xét tiếp vị trí i tiếp theo.

- Ngược lại, nếu gặp $a[k] >= a[i]$, tức tồn tại 1 số xếp không đúng vị trí, ta phải cho $a[i]$ về phía trước sao cho mảng được xếp đúng từ 0 đến i, bằng cách chèn $a[i]$ vào vị trí chính xác sao theo thứ tự, cụ thể: ví dụ 1 4 6 7 3 4 và ta đã xét đúng đến vị trí phần tử 7, xét $a[i] = 3$ và $a[k] = 7$, $a[k] > a[i]$ nên ta phải đổi $a[i]$ lên trước, dịch chuyển 4, 6, 7 về sau 1 đơn vị, chèn $a[i] = 3$ vào sau phần tử 1, kết thúc vòng lặp i này thu được 1 3 4 6 7 4.

Độ phức tạp: $O(N^2)$.

Code C++:

```
void insertionSort (vector<int> &a)
{
    for (int i = 1; i < a.size(); i++) {
        int e = a[i];
        int k;
        for (k = i - 1; k >= 0; k--) {
            if (a[k] < e) break;
            a[k + 1] = a[k];
        }
        a[k + 1] = e;
    }
}
```

Để thuật toán Stable (thứ tự các phần tử bằng nhau không đổi) thì đổi $a[k] <= e$.

2.5 Sắp xếp bằng phương pháp đóng gói (Shell Sort):

Ý tưởng:

- Đây là thuật toán biến thể, một phiên bản cải tiến của Insertion Sort, khi có thể tránh được việc vòng lặp thứ 2 của insertionSort phải thực hiện quá nhiều lần vì các số nhỏ đang nằm lệch về phía phải của dãy (thuật toán insertionSort phải thực hiện rất nhiều lần duyệt để dịch chuyển các phần tử làm cho độ phức tạp tăng lên);

- Tương tự cách cài tiến của CombSort so với BubbleSort (CombSort chia dãy cần sắp xếp thành nhiều dãy con và thực hiện BubbleSort trên dãy con đó), ta chia mảng thành nhiều dãy con và dùng insertionSort cho các dãy con đó;
- Cách chia: các phần tử nằm cách nhau 1 khoảng *interval* (*gap*) sẽ được xếp cùng một dãy con.

Độ phức tạp: $O(N^{1.25})$. Tuy có hẵn 3 vòng lặp, tuy nhiên số lần thực thi các vòng lặp đó nhỏ hơn rất nhiều so với độ lớn dữ liệu đầu vào N .

Code C++:

```
void shellSort (int *arr, int N)
{
    for (int interval = N/2; interval > 0; interval /= 2) {
        for (int i = interval; i < N; i++) {
            int e = *(arr + i);
            int j = i - interval;
            for (j; j >= 0; j -= interval) {
                if (*(arr + j) <= e) {
                    break;
                }
                *(arr + j + interval) = *(arr + j);
            }
            *(arr + j + interval) = e;
        }
    }
}
```

```
#include <iostream>
#include <ctime>
#include <cstdlib>

using namespace std;

void print(int *arr, const int N)
{
    for (int i = 0; i < N; ++i) {
        cout << *(arr + i) << " ";
    }
    cout << endl;
}

void insertionSort (int *arr, const int N)
{
```

```

for (int i = 1; i < N; ++i) {
    int e = *(arr+ i);
    int j;
    for (j = i - 1; j >= 0; --j) {
        if (*(arr + j) <= e) {
            break;
        }
        *(arr + j + 1) = *(arr + j);
    }
    *(arr + j + 1) = e;
}

void insertionSort1 (int *arr, int N)
{
    for (int i = 1; i < N; ++i) {
        for (int j = i - 1; j >= 0; --j) {
            if (*(arr + j) <= *(arr + j + 1)) {
                break;
            }
            swap(*(arr + j), *(arr + j + 1));
        }
    }
}

void insertionSort2(int *arr, const int N, int interval)
{
    for (int i = interval; i < N; i += interval) {
        for (int j = i - interval; j >= 0; j -= interval) {
            if (*(arr + j) < *(arr + j + interval)) {
                break;
            }
            swap(*(arr + j), *(arr + j + interval));
        }
    }
}

void shellSort(int *arr, const int N)
{
    for (int interval = N/2; interval > 2; interval /= 2) {
        for (int i = 0; i < interval; ++i) {
            insertionSort2(arr + i, N - i, interval);
        }
    }
    insertionSort2(arr, N, 1);
}

int main()

```

```

{
    const int N = 20;
    int *arr = new int[N];
    srand(time(NULL));
    for (int i = 0; i < N; ++i) {
        *(arr + i) = rand() % 30;
    }

    shellSort(arr, N);
    print(arr, N);

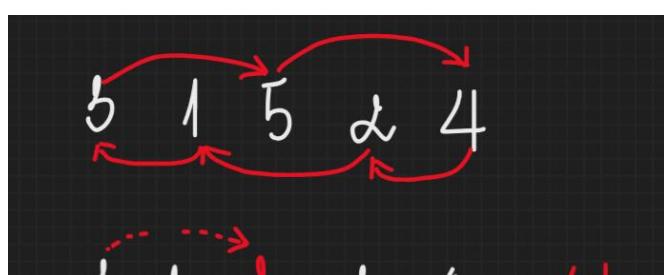
    delete[] arr;
}

```

2.6 Sắp xếp bằng phương pháp chu trình (Cycle Sort):

Ý tưởng:

- Thuật toán Cycle Sort là một thuật toán in-place (tại chỗ), tức hoán đổi trực tiếp trên chính mảng ban đầu;
- Được áp dụng khi yêu cầu quá trình thực thi không tạo ra quá nhiều lần hoán đổi (swap);
- Đây là thuật toán tối ưu về số lần hoán đổi (swap) về mặt lý thuyết. Mỗi phần tử chỉ được swap tối đa 01 lần: 00 lần nếu nó đã đúng vị trí, 01 lần nếu nó sai vị trí và swap vào vị trí đúng.
- Đây là thuật toán không tối ưu về độ phức tạp thời gian. Có thể lên đến $O(N^2)$.
- Ý tưởng:
 - + Duyệt qua $N - 1$ phần tử bằng vòng lặp for;
 - + Đặt phần tử đầu tiên vào đúng vị trí, bắt đầu chu trình;
 - + Nếu phần tử đầu tiên đã đúng vị trí, bắt đầu vòng lặp for tiếp theo (xét đến phần tử tiếp theo); Ngược lại, cho vị trí đúng của phần tử đầu tiên bằng giá trị của phần tử đầu tiên.
 - + Xét tương tự với phần tử tại vị trí bị thay thế cho đến khi không còn thay đổi được vị trí thêm.
- Mô phỏng:



Độ phức tạp: O(N^2)

Code C++:

```
#include <iostream>
#include <ctime>

using namespace std;

/* class CycleSort
{
    const int N = 12;
    int *arr;
public:
    CycleSort() {
        arr = new int [N];
        srand(time(NULL));
        for (int i = 0; i < N; i++) {
            arr[i] = rand() % 40;
        }
        cout << "Constructor create an array with N elements." << endl;
    }
    friend void print (CycleSort&);
    void cycleSort();
    ~CycleSort() {
        delete[] arr; arr = nullptr;
        cout << "Destructor removed all elements in arr." << endl;
    }
};

void print (CycleSort &CCS)
```

```

{
    for (int i = 0; i < CCS.N; i++) {
        cout << *(CCS.arr + i) << " ";
    }
    cout << endl;
} */

void CycleSort::cycleSort()
{
    for (int cycle_start = 0; cycle_start < N - 1; cycle_start++)
    {
        int item = arr[cycle_start];
        int pos = cycle_start;
        for (int i = cycle_start + 1; i < N; i++) {
            if (arr[i] < item) {
                pos++;
            }
        }
        if (pos == cycle_start) {
            continue;
        }
        // Nếu không có vòng while này, có khả năng gặp vòng lặp vô tận
        // Lấy ví dụ 3 1 3 1 1
        while (item == arr[pos]) {
            pos++;
        }
        if (item != arr[pos]) {
            swap(item, arr[pos]);
        }

        while (pos != cycle_start)
        {
            pos = cycle_start;
            for (int i = cycle_start + 1; i < N; i++) {
                if (arr[i] < item) {
                    pos++;
                }
            }
            while (item == arr[pos]) {
                pos++;
            }
            if (item != arr[pos]) {
                swap(item, arr[pos]);
            }
        }
    }
}
/*

```

```

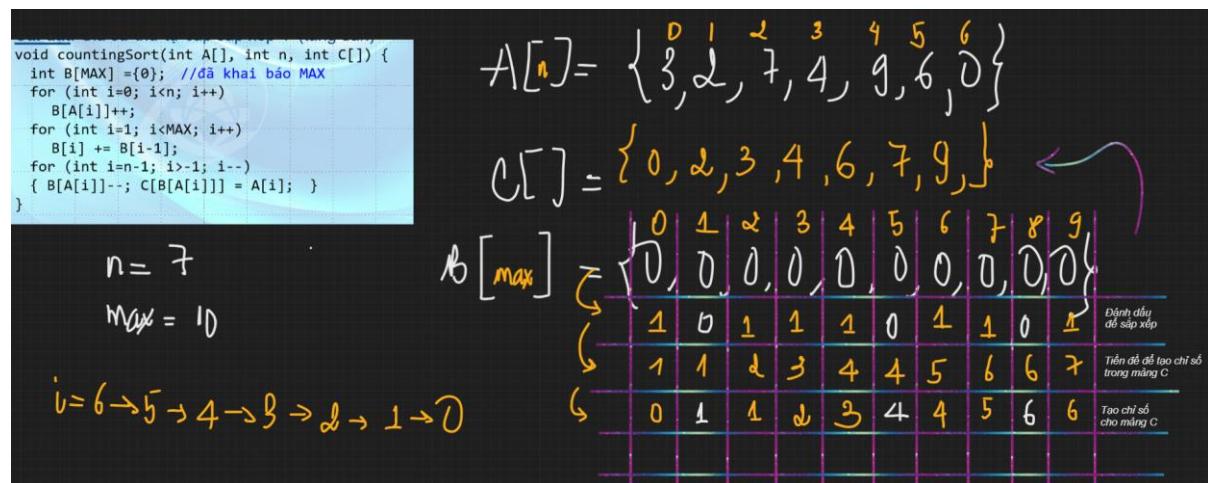
int main()
{
    CycleSort CCS;
    print(CCS);
    CCS.cycleSort();
    print(CCS);
} */

```

2.7 Sắp xếp bằng phương pháp đếm (Counting Sort):

Ý tưởng: Với mảng đầu vào a, tạo mảng đầu ra c và một mảng trung gian b. Đây là phương pháp sử dụng trong trường hợp mảng đầu vào a có phần tử lớn nhất không quá lớn, vì tư tưởng của thuật toán này là sắp xếp theo thứ tự số tự nhiên (được làm rõ ngay phía sau), nếu phần tử lớn nhất có giá trị quá lớn hoặc cách biệt quá lớn so với các phần tử còn lại thì mảng trung gian b có dung lượng cực kì lớn, làm giảm hiệu năng của thuật toán.

- Mảng trung gian b có MAX phần tử (MAX = phần tử có giá trị lớn nhất + 1), mỗi phần tử có giá trị tối đa là MAX – 1; đóng vai trò tạo chỉ số cho mảng đầu ra và sắp xếp các giá trị mảng đầu vào ứng với các chỉ số đó;
- Mảng đầu ra c có kích thước bằng mảng a và đã được sắp xếp thông qua mảng trung gian b;



Độ phức tạp: $O(N + MAX)$; (N là kích thước mảng a và c, MAX là kích thước mảng b).

Code C++:

```
/*
vector<int> a{1, 3, 5, 2, 0, 9, 12, 5, 2, 4};
vector<int> c(a.size(), 0);
const int MAX = 13 // Phần tử lớn nhất trong mảng + 1
//Đáp án là vector c với các phần tử đã được sắp xếp
*/

void countingSort(vector<int> &a, const int MAX, vector<int> &c)
{
    vector<int> b (MAX, 0);
    const int N = a.size();
    for (int i = 0; i < N; i++) {
        b[a[i]]++;
    }
    for (int i = 1; i < MAX; i++) {
        b[i] += b[i - 1];
    }
    for (int i = N - 1; i >= 0; i--) {
        b[a[i]]--;
        c[b[a[i]]] = a[i];
    }
}
```

Nhận xét: Chỉ nên dùng khi phần tử lớn nhất của mảng có giá trị không quá lớn tránh giảm hiệu suất và tránh tăng độ phức tạp.

Thuật toán Counting Sort luôn Stable khi vòng lặp cuối xét **từ cuối → đầu**. Phần tử nào đứng sau trong mảng ban đầu sẽ được phân bổ vào chỉ số lớn hơn (phía sau) → giữ nguyên thứ tự các phần tử bằng nhau trong mảng ban đầu. Khi xét **đầu → cuối**, thuật toán **không** đảm bảo tính stable.

2.8 Sắp xếp bằng phương pháp cơ sở (Radix Sort):

Ý tưởng: Phương pháp Radix Sort sẽ sắp xếp m lần (m là số chữ số tối đa của 1 số, ví dụ 321 có 3 chữ số thì m = 3), mỗi lần ứng với một vòng lặp for, mỗi vòng lặp for thực hiện việc sắp xếp các chữ số ở từng hàng một theo thứ tự (lưu ý “hàng” trong “hàng đơn vị”, “hàng chục”, “hàng trăm”,...).

- Bằng cách tái sử dụng chức năng của Counting Sort, hàm Radix Sort ngắn gọn hơn rất nhiều, do đó Counting Sort có thể xem như linh hồn của thuật toán này.

- Ví dụ:

- + 431 321 2 38 2943;
- + 0431 0321 0002 0038 2943; → chuyển 4 chữ số để xem
- + 0431 0321 0002 2943 0008; → so sánh hàng đơn vị
- + 0002 0008 0321 0431 2943; → so sánh hàng chục
- + 0002 0008 0321 0431 2943; → so sánh hàng trăm
- + 0002 0008 0321 0431 2943; → so sánh hàng nghìn
- + **Thuật toán kết thúc khi đã so sánh đến hàng nghìn, do số khác 0 ở bit lớn nhất xuất hiện đầu tiên ở bit hàng nghìn khi xét từ trái sang.**

Độ phức tạp: O(m.N) (m là số bit tối đa của phần tử lớn nhất trong mảng, N là số phần tử của mảng). Lý do: lặp m lần, mỗi lần thực hiện vòng lặp so sánh tồn N lần → m.N lần → O(m.N)

Code C++:

```
int getMax (int a[], int size)
{
    int max = a[0];
    for (int i = 1; i < size; i++) {
        max = (max < a[i]) ? a[i] : max;
    }
    return max;
}

void countingSort1 (int a[], int size, int place, int max)
{
    int indexArr[10] = {0};
    int output[size];

    for (int i = 0; i < size; i++) {
        indexArr[(a[i] / place) % 10]++;
    }
    for (int i = 1; i < 10; i++) {
        indexArr[i] += indexArr[i - 1];
    }
    for (int i = size - 1; i >= 0; i--) {
        indexArr[(a[i] / place) % 10]--;
        output[indexArr[(a[i] / place) % 10]] = a[i];
    }
}
```

```

        for (int i = 0; i < size; i++) {
            a[i] = output[i];
        }
    }

void radixSort (int a[], int size)
{
    int max = getMax(a, size);
    for (int place = 1; max / place > 0; place *= 10)
    {
        countingSort1 (a, size, place, max);
    }
}

```

Lưu ý: Khi sử dụng countingSort để phục vụ cho radixSort, cần đảm bảo countingSort phải stable, nếu không trường hợp các số có cùng số lượng chữ số rất có khả năng không sắp xếp đúng theo thứ tự mong muốn.

Ví dụ:

002 345 079 053 146

002 053 345 146 079

Nếu như không đảm bảo tính stable, thì khi xét đến chữ số thứ 3, bao gồm có: 0, 0, 3, 1, 0 lần lượt của các số (**002, 053, 345, 146, 079**), thứ tự của 3 chữ số 0 có thể bị thay đổi, dẫn đến kết quả sai. Điều cần làm là giữ nguyên thứ tự của 3 chữ số 0 đó, đồng nghĩa với việc thuật toán sắp xếp ở lần này phải stable.

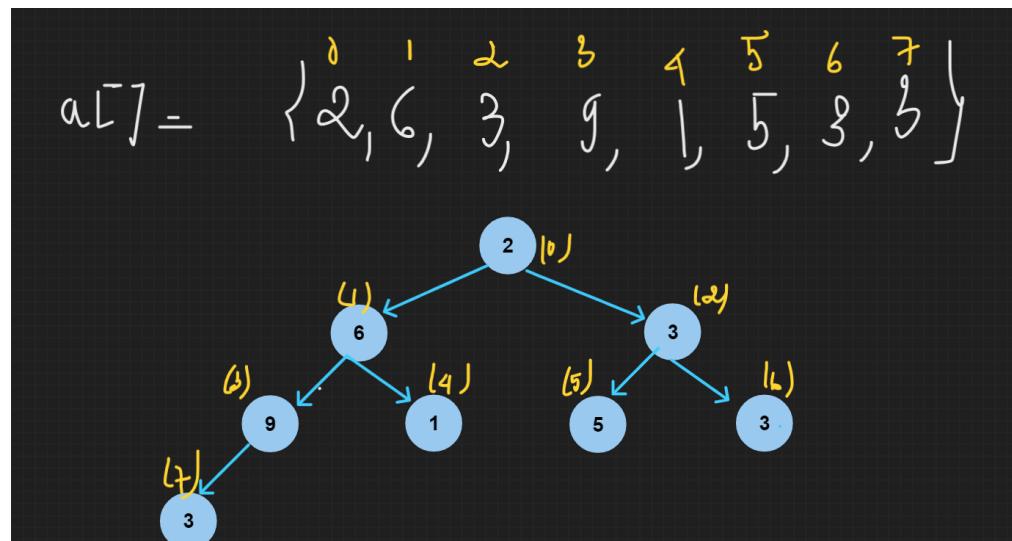
2.9 Sắp xếp bằng phương pháp vun đống (Heap Sort):

Ý tưởng:

- Cấu trúc Heap bản chất là một cây nhị phân hoàn chỉnh (complete binary tree), Heap có thể chia làm 2 loại: Max Heap (mọi nút cha đều lớn hơn hoặc bằng 2 nút con liền kề) và Min Heap (mọi nút cha đều bé hơn hoặc bằng 2 nút con liền kề).
- Một trong các thao tác cơ bản và chủ yếu của cấu trúc dữ liệu được khai triển Heap là thao tác Heapify. Độ phức tạp của thao tác này là $O(\log N)$ (N là số nút hiện có trong cây hay là số phần tử hiện có trong mảng), vì

số lần thực hiện các câu lệnh $O(1)$ chính bằng chiều cao của cây nhị phân hoàn chỉnh. Ta có: “There are at most m^h leaves in an m -ary tree of height h ”, như vậy có nhiều nhất 2^h nút trong cây nhị phân hoàn chỉnh đang xét (h là chiều cao), gọi N là số nút (số phần tử hiện có trong mảng), đồng nghĩa với $2^h = N$ hay $h = \log(N)$. Ta đã lí giải xong độ phức tạp của thao tác heapify.

- Ngoài ra, ta có thể hiểu theo cách sau: để tính được độ phức tạp, ta cần tính số lần thực thi câu lệnh $O(1)$. Thao tác Heapify hình thành dựa trên giải thuật đệ quy, tại mỗi nút được heapify, lệnh đệ quy gọi tới 1 trong 2 nút con của nó, rồi đệ quy tiếp đến 1 trong 2 nút con của nó, ... tiếp tục như vậy cho đến khi nút con cuối cùng là 1 lá. Nhận xét rằng: sau mỗi lần đệ quy, số lượng phần tử cần xét giảm một nửa, bài toán trở về tính độ phức tạp của lệnh đệ quy với N phần tử ban đầu, đệ quy nhiều lần với $N/2$. Do đó ta có: $N = 2^m$, với m là số lần gọi đệ quy, cũng chính là số thao tác $O(1)$ được thực thi. Như vậy, độ phức tạp sẽ là $m = \log(N)$ với N là độ lớn dữ liệu đầu vào hay số nút hiện có trong cây.
- Mỗi quan hệ giữa cây nhị phân và một mảng: nút cha có chỉ số là i thì 2 nút con trái và phải (nếu có) sẽ có chỉ số lần lượt là $2i + 1$ và $2i + 2$:



- **Đây là một phiên bản cải tiến của Selection Sort** vì đã tận dụng được kết quả của các phép so sánh trước đó, giảm độ phức tạp từ $O(n^2)$ xuống còn $O(n \log n)$.
- Cấu trúc chung:

- + *Hàm Heapify()*: sắp xếp mảng sao cho ở dạng cấu trúc Heap nó là max Heap, tức là phần tử đầu tiên luôn lớn nhất, xu hướng giảm dần từ nút cha đến các lá;
- + *Hàm HeapSort()*:
 - *Tạo maxHeap* bằng cách heapify tại tất cả các đỉnh trong của cây nhị phân hoàn chỉnh. Lưu ý: xét cuối → đầu.
 - *Lặp*:
 - Swap(giá trị lớn nhất (gốc), lá cuối cùng);
 - Bỏ qua lá cuối cùng, heapify tất cả các đỉnh còn lại. Lưu ý: xét đầu → cuối.
 - Lặp cho đến khi chỉ còn nút cha, khi đó mảng đã được sắp xếp hoàn chỉnh.
- Giải thích vì 2 vị trí gạch dưới:
- + Vị trí 1: cuối → đầu, cụ thể từ $(N/2 - 1) \rightarrow 0$ khi chiều đó đảm bảo các phần tử từ gốc đến lá đều mang xu hướng giảm dần, thứ tự ngược lại không đảm bảo điều đó và thường là sai.
- + Vị trí 2: đầu → cuối, vì sau khi đã tạo được một maxHeap, ta đã có một thứ tự sẵn, không cần duyệt tất cả các đỉnh trong nữa, chỉ duyệt lại những đỉnh bị thay đổi do vòng lặp gây ra, nhằm giảm thiểu chi phí thực thi bài toán. Hơn nữa, việc duyệt từ đầu → cuối trong trường hợp này cũng khiến cho việc truyền tham số vào hàm Heapify() dễ hơn (chỉ cần truyền số 0).

Độ phức tạp: $O(N\log N)$, hàm HeapSort() có 2 vòng lặp for và đều tốn $O(N\log N)$ do duyệt N lần hàm Heapify() với độ phức tạp $O(\log N)$.

Code C++:

```
#include <iostream>
#include <cstdlib> // srand() vs rand()
#include <ctime> // time()

using namespace std;

void heapify (int *arr, int N, int i)
{
```

```

int largest = i;
int left = 2*i + 1;
int right = 2*i + 2;

if (left < N && *(arr + left) > *(arr + largest)) {
    largest = left;
}
if (right < N && *(arr + right) > *(arr + largest)) {
    largest = right;
}

if (largest != i) {
    swap(*(arr + largest), *(arr + i));
    heapify(arr, N, largest);
}
}

void heapSort (int *arr, int N)
{
    for (int i = N/2 - 1; i >= 0; i--) {
        heapify(arr, N, i);
    }
    for (int i = N - 1; i >= 0; i--) {
        swap(*arr, *(arr + i));
        heapify(arr, i, 0);
    }
}

int main()
{
    const int N = 200;
    int *arr = new int [N];

    srand(time(NULL));
    for (int i = 0; i < N; i++) {
        *(arr + i) = rand() % 500;
    }

    heapSort(arr, N);
    for (int i = 0; i < N; i++) {
        cout << *(arr + i) << " ";
    }
    return 0;
}

```

2.10 Sắp xếp bằng phương pháp nhóm (xô) (Bucket Sort):

Ý tưởng:

- Thuật toán được sử dụng để sắp xếp 1 dãy các số nguyên hoặc số thực với số lượng phần tử không quá lớn, **phân bố đều** trên dãy. Lý do sẽ được giải thích ở đâu – thứ 3;
- Cách thực hiện:
 - + Chia dãy cần sắp xếp thành nhiều nhóm. Thứ tự các nhóm cũng tuân theo quy tắc sắp xếp ($>$ hay $<$);
 - + Sắp xếp các phần tử trong từng nhóm bằng các thuật toán khác (thông thường sử dụng hàm **sort** trong thư viện `<algorithm>`);
 - + Gộp các nhóm lại, thu được một dãy đã được sắp xếp, cũng chính là kết quả cần tìm.
- Lí do cần một dãy với các phần tử phân bố đều: Thuật toán được tạo ra với mục đích khai thác tối đa lợi ích của các nhóm (các xô – buckets) giúp chia nhỏ dãy ban đầu thành những dãy con ngắn hơn. Từ đó việc sắp xếp các dãy con đó cũng mang lại hiệu quả tốt hơn. Do đó, nếu đối mặt với dãy ban đầu không phân bố đều, tức các phần tử bị lệch về một (vài) phía nào đó, các nhóm (xô) không còn ý nghĩa, thuật toán khi đó cũng không có điểm khác biệt so với chỉ sử dụng hàm **sort** thông thường.
- Vấn đề khó khăn của thuật toán này chính là việc chọn **b_index** (xem code), tức cần xác định số lượng hộp sao cho hợp lý nhất. Nếu chia quá nhiều hộp hoặc quá ít hộp, có thể sẽ gây lãng phí bộ nhớ hoặc không đủ hộp để chia các phần tử đúng cách. Dù vậy, nguyên tắc chung cho việc chọn **b_index** là đảm bảo tính tuyến tính, để chắc chắn rằng ta có thể bảo tồn được tính chất đồng đều của dãy số ban đầu.

Độ phức tạp:

- $O(N.k)$ với k là số lượng phần tử trong mỗi dãy con (thông thường là số nguyên) do đó có thể xem như $O(N)$ – Viễn cảnh này chỉ đúng khi dãy ban đầu *phân bố đồng đều* và *số lượng phần tử ban đầu đủ nhỏ* để có thể xem k nhỏ hơn rất nhiều so với N .

- Khi dãy không phân bố đều, k có thể bằng N, độ phức tạp nhảy vọt lên đến $O(N^2)$.

Code C++:

```

/* class BucketSort
{
    const int N = 15;
    vector<float> arr;
public:
    BucketSort() : arr(N) {
        srand(time(NULL));
        for (int i = 0; i < N; i++) {
            arr[i] = ((rand() % 1000) * 1.0)/1000;
        }
    }
    void buckketSort();
    friend void print (BucketSort&);
}; */

void BucketSort::buckketSort()
{
    vector<float> b[N];

    int b_index;
    for (int i = 0; i < N; i++) {
        b_index = N * arr[i];
        b[b_index].push_back(arr[i]);
    }

    for (int i = 0; i < N; i++) {
        sort(b[i].begin(), b[i].end(), less<int>());
    }

    int index = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < b[i].size(); j++) {
            arr[index++] = b[i][j];
        }
    }
}

/* void print (BucketSort &BKS)
{
    for (float i : BKS.arr) {
        cout << i << " ";
    }
    cout << endl;
}

```

```

}

int main()
{
    BucketSort BKS;
    print(BKS);
    BKS.bucketSort();
    print(BKS);
} */

```

Cách không sử dụng priority_queue: Như ta đã biết, hàng đợi ưu tiên có tác dụng thay thế các thuật toán sắp xếp khi tần suất sắp xếp quá nhiều lần. Với lợi thế và độ phức tạp tuyến tính, hàng đợi ưu tiên luôn là lựa chọn tốt cho sắp xếp nhiều lần.

```

void BucketSort::bucketSort()
{
    vector<priority_queue<float, vector<float>, greater<float>>> b(N);

    int b_index;
    for (int i = 0; i < N; i++) {
        b_index = N * arr[i];
        b[b_index].push(arr[i]);
    }

    int index = 0;
    for (int i = 0; i < N; i++) {
        while (!b[i].empty()) {
            arr[index++] = b[i].top();
            b[i].pop();
        }
    }
}

```

2.11 Sắp xếp bằng phương pháp phân hoạch (Quick Sort):

Đã được học tại [đây](#).

2.12 Sắp xếp bằng phương pháp trộn (Merge Sort):

Đã được học tại [đây](#).

Nhận xét chung về các thuật toán sắp xếp trên:

- Các thuật toán sắp xếp có tính Stable:
 - + Counting Sort: *khi vòng lặp cuối duyệt cuối* \rightarrow *đầu*
 - + Radix Sort (vì chủ chốt vẫn sắp xếp dựa trên Counting Sort);
 - + (Selection Sort): *khi điều kiện dừng vòng lặp có “=”*
 - + (Bubble Sort): *khi điều kiện swap có dấu “=”*
 - + (Insertion Sort): *khi điều kiện dừng vòng lặp có “=”*
- Các thuật toán sắp xếp không có tính Stable:
 - + Selection Sort: *khi điều kiện dừng vòng lặp không có “=”*
 - + (Bubble Sort): *khi điều kiện swap không có dấu “=”*
 - + Insertion Sort: *khi điều kiện dừng vòng lặp không có “=”*
 - + (Counting Sort): *khi vòng lặp cuối duyệt đầu* \rightarrow *cuối*
 - + (Radix Sort): *khi vòng lặp cuối duyệt đầu* \rightarrow *cuối*
 - + Heap Sort.

C. CÁC CẤU TRÚC DỮ LIỆU CHÍNH:

Bức tranh chung:

Nhắc lại: cấu trúc dữ liệu là một tập hợp các dữ liệu được sắp xếp theo một trật tự cụ thể, là cách tổ chức và lưu trữ dữ liệu trong bộ nhớ của máy tính sao cho có thể sử dụng chúng một cách hiệu quả.

Tiêu chí so sánh	CTDL được khai triển (Concrete Data Structure)	CTLD trừu tượng (Abstract Data Structure)
1. Khái niệm	Là một <i>cách cụ thể</i> để lưu trữ và tổ chức dữ liệu trong bộ nhớ của máy tính.	Là một <i>mô hình toán học</i> mô tả: cách dữ liệu được lưu trữ và tổ chức + các phép toán được thực hiện trên nó.
2. Tính chất	<ul style="list-style-type: none">- Xác định cụ thể: cách dữ liệu được tổ chức, các phép toán được thực hiện, các quy tắc ràng buộc, ...- Được triển khai bằng các dữ liệu cơ bản (mảng, DSLK, cây, heap, ...).→ Phụ thuộc vào ngôn ngữ và môi trường lập trình.	<ul style="list-style-type: none">- Mô tả tính chất chung: các phép toán và quy tắc trừu tượng, ...- Không được triển khai cụ thể về cách tổ chức dữ liệu.→ Độc lập với ngôn ngữ và môi trường lập trình.
3. Ví dụ	<ul style="list-style-type: none">- Mảng (array)- DSLK đơn (singly linked list)- Cây nhị phân (Binary Tree)- ...	<ul style="list-style-type: none">- Danh sách trừu tượng (Abstract List)- Hàng đợi trừu tượng (Abstract Queue)- Cấu trúc dữ liệu trừu tượng cho Đồ thị (Graph)- ...

I. CẤU TRÚC DỮ LIỆU NGĂN XÉP (STACK):

1. Tổng quan:

Là một cấu trúc dữ liệu trừu tượng, tức đây là một cấu trúc dữ liệu không phụ thuộc vào ngôn ngữ cũng như môi trường lập trình. Trong các môi trường lập trình khác nhau, chúng đều hỗ trợ các thao tác với chức năng như nhau mà không quan tâm đến cấu trúc dữ liệu đằng sau. (Tuy nhiên có thể khác nhau về cú pháp).

Đặc điểm cốt lõi của "ngăn xếp" là bạn chỉ có thể thêm và xóa các phần tử từ cùng một đầu. Việc bạn triển khai nó như thế nào không quan trọng bởi "ngăn xếp" chỉ là một giao diện trừu tượng.

Ngăn xếp và đệ quy rất giống nhau. Nguyên nhân là do đệ quy được thực hiện bằng cách sử dụng ngăn xếp. Các lệnh gọi hàm được đẩy vào ngăn xếp và trên mỗi câu lệnh trả về, lệnh gọi hiện tại được đưa ra khỏi ngăn xếp. Đỉnh của ngăn xếp tại bất kỳ thời điểm nào là lệnh gọi hàm hiện tại.

Các dạng cấu trúc dữ liệu ngăn xếp:

1. Ngăn xếp có kích thước cố định;
2. Ngăn xếp có kích thước động;
3. Ngăn xếp tiền tố thành hậu tố;
4. Ngăn xếp đánh giá biểu thức;
5. Ngăn xếp đệ quy;
6. Ngăn xếp quản lý bộ nhớ
7. Ngăn xếp dấu ngoặc cân bằng;
8. Undo – Redo Stack;

2. Các dạng toán thường sử dụng cấu trúc stack:

Nhìn chung, các bài toán mang hơi hướng của tính chất LIFO đều áp dụng được cấu trúc dữ liệu stack để giải quyết. Tuy nhiên để nhận biết dấu hiệu này không phải là điều đơn giản.

Một số bài toán cơ bản có ứng dụng stack:

1. Đảo ngược chuỗi (kí tự): LIFO thể hiện ở việc từ nào trong chuỗi (kí tự nào trong từ) được đọc cuối sẽ được in ra đầu tiên;

2. Chuyển đổi prefix – infix – postfix;
3. Các thao tác (nút “quay lại”) trên các web: khi mở một URL mới, nó sẽ được lưu vào một cấu trúc dữ liệu stack (đại diện là nút “Quay lại”, tức nó sẽ lưu tất cả các URL mà ta đã truy cập). Mỗi lần truy cập vào URL mới, nó sẽ được lưu vào đỉnh của stack, khi nhấn nút quay lại, URL hiện tại sẽ được xóa khỏi đỉnh stack, đỉnh stack hiện tại sẽ là URL được truy cập ở lần gần nhất trước đó.

3. Các ví dụ điển hình:

Ví dụ 1:

Cho một chuỗi (biểu thức toán học), kiểm tra xem các cặp dấu ngoặc đơn có hợp lệ không?

Ví dụ: Cho các chuỗi (biểu thức) sau: $(x + 4) - (4 + y)$; $(x - 5)) + (r)^((t + 4)$. Đáp án sẽ trả về TRUE cho biểu thức đầu tiên, và trả về FALSE cho biểu thức còn lại.*

Lời giải:

Ý tưởng:

- Duyệt từ đầu → cuối chuỗi (chỉ quan tâm những vị trí là ‘(’ hoặc ’)'). Nếu là ‘(’ thì bỏ vào stack, gặp ’)’ thì xem trong stack có rỗng hay không, nếu có (tức chỉ có ’)’ mà không có ‘(’ thì lập tức trả về FALSE. Ngược lại, nếu stack không rỗng, thì xóa ‘(’ trên đỉnh stack, khi đó 1 cặp ngoặc đơn hợp lệ được duyệt, tiếp tục quá trình đến khi hết chuỗi.
- Sau khi kết thúc chuỗi, nếu stack vẫn chưa rỗng, tức tồn tại ‘(’ mà không có ’)', khi đó lập tức trả về FALSE.
- Bài toán kết thúc.

Độ phức tạp: O(N) với N là độ dài chuỗi đã cho.

Code C++:

```
bool check (string s)
{
    stack<char> st;
```

```

int N = s.length();
for (char i : s) {
    if (i == '(') {
        st.push(i);
    }
    else if (i == ')') {
        if (st.empty()) {
            return false;
        }
        else {
            st.pop();
        }
    }
}
if (!st.empty()) return false;
return true;
}

int main ()
{
    ifstream in ("Input.txt");
    string s;
    getline (in, s);
    cout << check(s);
}

```

```

bool check1 (stack<char> &st, string str) {
    if (str[0] == '\0') {
        return st.empty();
    }
    else if (str[0] == '(') {
        st.push(str[0]);
    }
    else if (str[0] == ')') {
        if (st.empty() == true) {
            return false;
        }
        else {
            st.pop();
        }
    }

    return check1(st, str.substr(1));
}

```

Ví dụ 2:

Tìm phần tử lớn hơn đầu tiên bên phải (find next greater elements) 

Ví dụ: cho 1 mảng $arr[5] = \{5, 6, 2, 5, 1\}$ thì:

- Phần tử lớn hơn 5 đầu tiên bên phải là 6;
- Phần tử lớn hơn 6 đầu tiên bên phải là KHÔNG CÓ, trả về -1;
- Phần tử lớn hơn 2 đầu tiên bên phải là 5;
- Phần tử lớn hơn 5 đầu tiên bên phải là KHÔNG CÓ, trả về -1;
- Phần tử lớn hơn 1 đầu tiên bên phải là KHÔNG CÓ, trả về -1;

Như vậy, kết quả trả về sẽ là một mảng có kích thước bằng kích thước của mảng ban đầu arr , và lần lượt chứa các phần tử: 6, -1, 5, -1, -1;

Lời giải:

Ý tưởng:

- Duyệt qua từng phần tử trong mảng bằng cách duyệt từng **chỉ số**.
- Push chỉ số i vào stack, kiểm tra xem phần tử tại i + 1 có lớn hơn hoặc bằng phần tử tại chỉ số i hay không:
 - + Nếu có, gán cho mảng đầu ra tại vị trí i giá trị tại i + 1 (vì khi đó giá trị tại i + 1 lúc này chính là phần tử đầu tiên lớn hơn bên phải cần tìm), sau đó lấy i ra khỏi stack;
 - + Nếu không, push i + 1 vào stack (lúc này i + 1 nằm trên i trong stack);
 - + Tuy nhiên, hành động này chỉ có thể so sánh i và i + 1, để cải tiến thành một phương pháp làm, ta xem đầu dòng ngay sau đây.
- Push chỉ số i vào stack (i là đỉnh stack), kiểm tra xem phần tử tại i + 1 có lớn hơn hoặc bằng phần tử tại chỉ số <giá trị trên đỉnh stack> hay không:
 - + Nếu có, gán cho mảng đầu ra tại vị trí <giá trị trên đỉnh stack> giá trị tại i + 1 (vì khi đó giá trị tại i + 1 lúc này chính là phần tử đầu tiên lớn hơn bên phải cần tìm), sau đó lấy <giá trị trên đỉnh stack> ra khỏi stack; sau đó push i + 1 vào stack.
 - + Nếu không, push i + 1 vào stack (lúc này i + 1 nằm trên <giá trị trên đỉnh stack> trong stack và trở thành đỉnh stack mới);

- + Lặp lại hành động này cho đến khi stack trở nên rỗng hoặc duyệt hết tất cả các phần tử.
- Trong trường hợp đã duyệt hết tất cả phần tử mà stack vẫn còn, gán cho mảng đầu ra tại vị trí <giá trị trên đỉnh stack> giá trị -1 và pop đỉnh stack, cho đến khi stack trở nên rỗng.

Độ phức tạp: $O(N^2)$, vì vòng while phía trong for có độ phức tạp trong trường hợp xấu nhất là N .

Code C++:

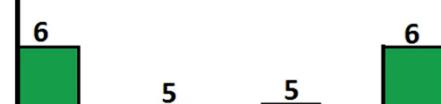
```
int* findMax (int *arr, const int N)
{
    int *output = new int [N];
    stack<int> s;
    for (int i = 0; i < N; i++) {
        if (s.empty()) {
            s.push(i);
        }
        else {
            while (!s.empty() && (*arr + s.top()) <= *(arr + i)) {
                *(output + s.top()) = *(arr + i);
                s.pop();
            }
            s.push(i);
        }
    }
    while (!s.empty()) {
        *(output + s.top()) = -1;
        s.pop();
    }
    return output;
}
```

Ví dụ 3:

Tìm hình chữ nhật có diện tích lớn nhất được tạo thành từ các cột của một biểu đồ tần suất histogram (find max Rectangular Area in a histogram).

Ví dụ cho một histogram như sau:

Max Area = 3 x 4 = 12



Diện tích lớn nhất có thể tìm thấy sẽ được tạo từ 3 cột 2, 3, 4 (chiều cao tương ứng là 5, 4, 5).

Lời giải:

Ý tưởng:

- Tìm tất cả diện tích hình chữ nhật cần thiết, sau đó chọn giá trị lớn nhất;
- Xem các giá trị của mỗi cột là một phần tử trong mảng có chỉ số bắt đầu từ 0, trong ví dụ của đề bài giả sử mảng đó tên **arr**;
- Duyệt qua từng phần tử trong mảng, *ứng với mỗi phần tử, tìm diện tích hình chữ nhật lớn nhất có thể tồn tại* (với chiều cao là giá trị của phần tử đó). Tức là xem tại vị trí i , với chiều cao h , thì diện tích hình chữ nhật lớn nhất có thể có là gì. Như vậy sau khi xét qua tất cả các giá trị i (duyệt toàn bộ mảng), ta thu được giá trị **max** cần tìm;
- Phương pháp:
 - + Đưa chỉ số phần tử đầu tiên vào stack (sau này khi stack rỗng, ta buộc phải đưa chỉ số hiện tại vào), xét tiếp các phần tử tiếp theo (tổng quát phần tử đó có chỉ số là i);
 - + Nếu phần tử tại i lớn hơn hoặc bằng phần tử tại *<giá trị trên đỉnh stack>* (tức tìm được một cột nào đó cao hơn hoặc bằng), thì push i vào stack. Vì khi tồn tại cột sau cao hơn, đồng nghĩa với việc hình chữ nhật với chiều cao bằng chiều cao của các cột trước đó còn có thể mở rộng, lúc đó ta chưa vội tính diện tích;
 - + Ngược lại, nếu phần tử tại i nhỏ hơn phần tử tại *<giá trị trên đỉnh stack>* (tức tìm được một cột nào đó thấp hơn), thì tính diện tích hình chữ nhật có thể có tại tất cả các đỉnh cao hơn trước đó. Vì khi tồn tại cột sau thấp

- hơn, đồng nghĩa với việc diện tích hình chữ nhật với chiều cao bằng chiều cao của các cột trước đó không thể mở rộng, lúc đó ta sẽ tính diện tích;
- + Lặp lại quá trình cho đến khi tất cả phần tử của mảng được duyệt.
 - + **Nhận xét:** stack chỉ chứa chỉ số của các phần tử với giá trị tăng dần, vì ta chỉ thêm chỉ số i vào đỉnh nếu phần tử tại i lớn hơn (=) phần tử trên đỉnh hiện tại, hay nói cách khác, ta chỉ thêm vào stack nếu cột sau cao hơn (=) cột trước. Ngoài ra, vòng lặp chỉ nên dùng while() thay vì for() vì chỉ số i không phải lúc nào cũng được tăng lên sau mỗi vòng lặp. Nếu muốn dùng for() thì xem nhận xét sau phần code.
 - Sau đó, ta cần tính diện tích tại các cột còn tồn đọng trong stack. Xử lý tương tự cho đến khi stack rỗng. Nếu không có phần này, sẽ dẫn đến kết quả sai, ta xem ví dụ trong đề bài:
 - + Sau khi duyệt hết các phần tử trong mảng đầu vào, stack hiện tại ở trạng thái như sau:
 - + Giả sử cột số cuối cùng (chỉ số 6) không phải giá trị 6 nữa, mà là 100, thì ta đã bỏ qua diện tích lớn nhất đáng có. Nếu chạy chương trình, diện tích lớn nhất lúc này vẫn là 12 thay vì 100 như mong đợi.
 - + Vì vậy, cần phải xét toàn bộ các phần tử còn sót lại trong stack như khi đang duyệt các phần tử trong mảng đầu vào, để tránh trường hợp sai sót không đáng có.

Độ phức tạp: O(N) (N là số cột của biểu đồ - số phần tử trong mảng)

Code C++:

```
int maxRectangularArea(int *arr, const int N)
{
    stack<int> st;
    int i = 0;
    int res = -1;
    while (i < N)
    {
        if (st.empty() || *(arr + i) >= *(arr + st.top()))
        {
            st.push(i);
        }
        else
        {
            st.pop();
            if (!st.empty())
                res = max(res, (i - st.top()) * *(arr + i));
        }
    }
}
```

```

        st.push(i);
        i++;
    }
    else // Tinh dien tich
    {
        int index = st.top();
        st.pop();
        if (st.empty())
        {
            res = max(res, *(arr + index) * i);
        }
        else
        {
            res = max(res, *(arr + index) * (i - st.top() - 1));
        }
    }
}
while (!st.empty())
{
    int index = st.top();
    st.pop();
    if (st.empty())
    {
        res = max(res, *(arr + index) * i);
    }
    else
    {
        res = max(res, *(arr + index) * (i - st.top() - 1));
    }
}
return res;
}

```

Nhận xét: Nếu muốn dùng for thay cho while, ta thực hiện như sau. Sự khác biệt diễn ra khi for tăng i liên tục, trong khi đó while chỉ tăng i khi ta chủ động tăng i. Để code chạy đúng, ta thêm “i--” như sau:

```

int solve (int *arr, const int N)
{
    stack<int> st;
    //int i = 0;
    int res = -1;
    for (int i = 0; i < N; i++)
    {
        if (st.empty() || *(arr + i) >= *(arr + st.top()))
        {
            st.push(i);
        }
        else
        {
            res = max(res, *(arr + st.top()) * (i - st.top() - 1));
            st.pop();
        }
    }
    return res;
}

```

```
//i++;
}
else {
    int index = st.top();
    st.pop();
    if (st.empty()) {
        res = max(res, *(arr + index) * i);
    }
    else {
        res = max(res, *(arr + index) * (i - st.top() - 1));
    }
    i--;
}
return res;
}
```

Ví dụ 4:

Tìm tất cả xâu con có k kí tự khác nhau trong một chuỗi s bất kỳ.

Ví dụ:

- $s = "abcaaabccb"$ và $k = 3$;
- Yêu cầu: tìm tất cả xâu con của s sao cho mỗi xâu con chứa 3 kí tự khác nhau (độ dài xâu con có thể khác 3).
- Kết quả: $abc, bca, caaab, aaabc, aabc, abc$.

Lời giải:

Ý tưởng:

- Cách 1: Áp dụng giải thuật ngây thơ tìm lần lượt tất cả các xâu con thỏa mãn điều kiện. Độ phức tạp: $O(N^2)$.
- Cách 2: Úng dụng cấu trúc dữ liệu stack.

Code C++:

Cách 1:

```
vector<string> findSubString(string s, int k)
{
    vector<string> res;
    int N = s.length();
    int count;
    string tmp;

    for (int i = 0; i < N; i++) {
        tmp = "";
        count = k;
        bool isUsed[26];
        for (int a = 0; a < 26; a++) {
            isUsed[a] = false;
        }
        for (int j = i; j < N + 1; j++) {
            if (count == 0) {
                res.push_back(tmp);
                break;
            }
            if (isUsed[s[j] - 'a'] == false) {
                count--;
            }
        }
    }
}
```

```

        tmp += s[j];
        isUsed[s[j] - 'a'] = true;
    }
}
return res;
}

```

Lưu ý: khi sử dụng cách 1, vòng lặp for con với chỉ số j phải duyệt từ i đến N, thay vì từ 0 đến N – 1 như vòng lặp for mẹ. Bởi vì nếu j chỉ duyệt đến N – 1 sẽ có thể làm mất cấu hình xâu con cuối cùng, do chúng ta chỉ kiểm tra điều kiện break khi j nhảy sang một giá trị mới, do đó sẽ thiếu xâu cuối cùng nếu j đã duyệt tới N – 1. Khi đó vòng lặp for j dừng trong khi cấu hình con cuối chưa được in ra.

Ngoài ra, ta có thể gán mảng isUsed[26] với các giá trị ban đầu bằng false thông qua hàm memset như sau.

```
memset(isUsed, false, sizeof(isUsed));
```

Cũng cần lưu ý, chỉ được dùng sizeof(isUsed), không được dùng sizeof(bool) vì isUsed là một mảng, kích thước (tính theo số byte) của hàm isUsed là 26 (vì mảng có 26 phần tử, mỗi phần tử boolean chiếm 1 byte), trong khi sizeof(bool) là 1 byte.

Ví dụ 5:

Giải mã tăng giảm

Cho mảng A[] chỉ bao gồm các ký tự I hoặc D. Ký tự I được hiểu là tăng (Increasing) ký tự. D được hiểu là giảm (Decreasing). Sử dụng các số từ 1 đến 9, hãy đưa ra số nhỏ nhất được đoán nhận từ mảng A[]. Chú ý, các số không được phép lặp lại. Dưới đây là một số ví dụ mẫu:

- *A[] = "I" : số tăng nhỏ nhất là 12.*
- *A[] = "D" : số giảm nhỏ nhất là 21*
- *A[] = "DD" : số giảm nhỏ nhất là 321*
- *A[] = "DDIDDDIID": số thỏa mãn 321654798*

Input:

- *Dòng đầu tiên đưa vào số lượng bộ test T.*
- *Những dòng kế tiếp đưa vào T bộ test. Mỗi bộ test là một xâu ID*

- $T, Length(A)$ thỏa mãn ràng buộc: $1 \leq T \leq 100; 1 \leq Length(A) \leq 9$; .

Output:

- *Đưa ra kết quả mỗi test theo từng dòng.*

Lời giải:

Ý tưởng:

- Duyệt qua từng kí tự trong chuỗi đã cho;
- Nếu kí tự đó là ‘I’ thì đưa vào hàm Process_I (Xử lí trường hợp I);
- Nếu kí tự đó là ‘D’ thì đưa vào hàm Process_D (Xử lí trường hợp D);
- Duy trì một vector **num** chứa giá trị từ 1 đến 9, mỗi lần xử lí, vector này sẽ được cập nhật (xóa bỏ phần tử, ...);
- Duy trì một vector **res** sẽ chứa mã được giải;
- Hàm Process_I:
 - o Gọi **count** là số kí tự I liên tiếp bắt đầu từ vị trí đang xét (không tính các kí tự I liền trước kí tự D);
 - o Gán **<count>** giá trị đầu tiên trong vector **num** vào vector **res** theo thứ tự xuôi (*tăng dần*);
 - o Sau đó xóa các giá trị đã được gán trong vector **num** đi;
- Hàm Process_D:
 - o Gọi **count** là số kí tự D liên tiếp bắt đầu từ vị trí đang xét;
 - o Gán **<count>** giá trị đầu tiên trong vector **num** vào vector **res** theo thứ tự ngược (*giảm dần*);
 - o Sau đó xóa các giá trị đã được gán trong vector **num** đi;

Độ phức tạp: Xấu nhất: $O(N^2)$ với N là độ dài chuỗi. Tốt nhất là

Code C++:

```
#include <iostream>
#include <vector>
#include <fstream>

using namespace std;

void print (vector<int> a)
```

```

{
    for (vector<int>::iterator i = a.begin(); i != a.end(); i++) {
        cout << *(i.base()) << " ";
    }
    cout << endl;
}

void Process_D (string s, int &i, vector<int> &num, vector<int> &res)
{
    int count = 0;
    while (s[i] == 'D') {
        count++;
        i++;
    }
    for (int j = count; j >= 0; j--) {
        res.push_back(num[j]);
    }
    num.erase(num.begin(), num.begin() + count + 1);
}

void Process_I (string s, int &i, vector<int> &num, vector<int> &res)
{
    int count = 0;
    while ((s[i] == 'I' && s[i + 1] != 'D' && s[i + 1] != '\0') || (s[i] == 'I' && s[i + 1] == '\0')) {
        count++;
        i++;
    }
    for (int j = 0; j <= count; j++) {
        res.push_back(num[j]);
    }
    num.erase(num.begin(), num.begin() + count + 1);
}

vector<int> decryption (string s)
{
    vector<int> num = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    vector<int> res;

    int N = s.length();
    for (int i = 0; i <= N; i++) {
        if (s[i] == 'I') {
            Process_I(s, i, num, res);
        }
        else {
            Process_D(s, i, num, res);
        }
    }
}

```

```
    return res;
}

int main()
{
    ifstream input;
    input.open("Input.txt");
    int num;
    input >> num;
    for (int i = 0; i < num; i++) {
        string s;
        input >> s;
        cout << s << endl << "\t";
        vector<int> res = decryption(s);
        print(res);
    }
}
```

Lưu ý: trong hàm decryption xét i từ 0 → N thay vì N – 1. Vì các hàm Process_D, Process_I đã tăng i lên 1 đơn vị thay cho vòng lặp for, nên ta phải xét đến N để tránh mất đi cấu hình.

II. CÂU TRÚC DỮ LIỆU HÀNG ĐỢI (QUEUE):

1. Tổng quan:

Là một cấu trúc dữ liệu trừu tượng, tức đây là một cấu trúc dữ liệu không phụ thuộc vào ngôn ngữ cũng như môi trường lập trình. Trong các môi trường lập trình khác nhau, chúng đều hỗ trợ các thao tác với chức năng như nhau mà không quan tâm đến cấu trúc dữ liệu đằng sau. (Tuy nhiên có thể khác nhau về cú pháp).

Hoạt động dựa trên nguyên tắc FIFO (first in first out).

2. Các dạng toán sử dụng hàng đợi (queue):

Hàng đợi là một cấu trúc dữ liệu được ưu tiên trong các bài toán sinh ra các cấu hình (số, chuỗi, ...) mà dựa vào các cấu hình đã có trước đó.

Đặc biệt với hàng đợi ưu tiên, nhờ có ưu thế về độ phức tạp mà đa số các lập trình viên dùng để thay thế các thuật toán sắp xếp khi phải sắp xếp dữ liệu với tần suất cao (như phải cập nhật thứ tự sau mỗi lần xử lý, ...).

3. Các bài toán điển hình:

a. Hàng đợi thông thường:

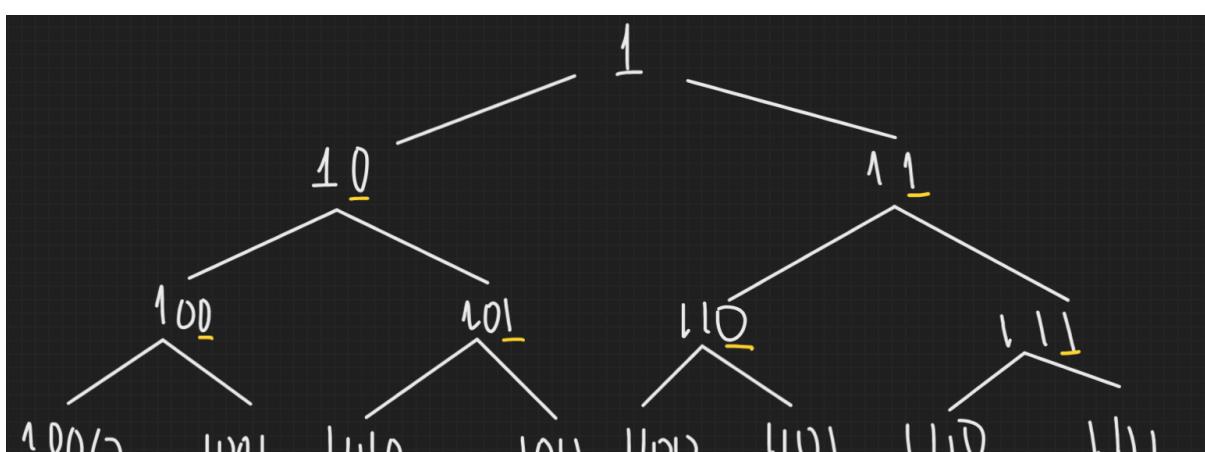
Ví dụ 1:

Bài toán sinh các chuỗi bit nhị phân từ 1 đến N (N hữu hạn cho trước)

Lời giải:

Ý tưởng:

- Push “1” vào queue;
- Lấy phần tử phía trước queue (“1”) ra, gán phía sau đó lần lượt “0” và “1” sẽ được 2 chuỗi mới là “10” và “11”, push theo thứ tự 2 chuỗi đó vào queue;
- Quá trình lặp lại cho tới khi in ra đủ N số.
- Mô phỏng:



Độ phức tạp: O(N) (với N là số chuỗi bit ban đầu)

Code C++:

```
void print (vector<string> s, int limit)
{
    for (int i = 0; i < limit; i++) {
        cout << s[i] << " ";
    }
    cout << endl;
}

vector<string> BinaryBit (void)
{
    vector<string> res;
    queue<string> q;
    q.push("1");
    res.push_back("1");
    while (res.size() < 100) {
        string s1 = q.front() + "0";
        string s2 = q.front() + "1";
        q.push(s1), q.push(s2);
        q.pop();
        res.push_back(s1), res.push_back(s2);
    }
    return res;
}
```

Lưu ý: Ưu điểm của lời giải áp dụng hàng đợi so với backTracking và Sinh tất cả câu hình chính là độ phức tạp thời gian ít hơn (tuyến tính so với mũ 2^N (quay lui)). Đây là một cải tiến vô cùng lớn.

Ví dụ 2:

Bài toán in ra bội số nhỏ nhất của một số nguyên dương N cho trước chỉ gồm các chữ số 0 và 9.

Ví dụ:

- $N = 4$ thì bội số thỏa mãn là 900;
- $N = 5$ thì bội số thỏa mãn là 90;

Lời giải:

Ý tưởng:

- Tương tự ví dụ 1 (sinh ra chuỗi nhị phân), ta cũng thực hiện việc in ra chuỗi chỉ bao gồm các chỉ số 0 và 9 (vai trò của 9 trong ví dụ này và vai trò của 1 trong ví dụ 1 là như nhau);
- Với mỗi chuỗi tìm được, chuyển chuỗi về số nguyên (nên có kiểu là long long thông qua hàm `stoll()`) và kiểm tra xem số đó có thỏa mãn việc chia hết cho N hay không, nếu có thì return ngay giá trị đó, ngược lại xét bội kế tiếp.
- Lưu ý: ngoài việc thực hiện gián tiếp, tức có sự chuyển đổi giữa chuỗi và số nguyên, thì ta có thể xét trực tiếp dưới dạng số nguyên ngay từ ban đầu. Khi đó queue sẽ chứa các giá trị nguyên thay vì chứa các chuỗi như ta đã làm ở ví dụ trước.

Độ phức tạp: O(N)

Code C++:

Trường hợp queue chứa các string:

```
long long LeastMultiplication (const int N)
{
    queue<string> q;
    q.push("9");
    while (true)
    {
        long long num1 = stoll(q.front() + "0");
        if (num1 % N == 0) {
            return num1;
        }
        long long num2 = stoll(q.front() + "9");
        if (num2 % N == 0) {
            return num2;
        }
        q.push(q.front() + "0");
        q.push(q.front() + "9");
        q.pop();
    }
}
```

```
}
```

Trường hợp queue chứa long long:

```
long long LeastMultiplication (const int N)
{
    queue<long long> q;
    q.push(9);
    while (true)
    {
        long long num1 = q.front() * 10;
        if (num1 % N == 0) {
            return num1;
        }
        long long num2 = q.front() * 10 + 9;
        if (num2 % N == 0) {
            return num2;
        }
        q.push(num1);
        q.push(num2);
        q.pop();
    }
}
```

Ví dụ 3:

Bài toán số lộc phát.

Số lộc phát là số được hình thành chỉ từ 2 chữ số 6 và 8. Với một số nguyên dương M cho trước ($M \leq 15$), hãy in ra tất cả các số lộc phát có tối đa M chữ số. Lưu ý in theo thứ tự từ lớn đến bé.

Lời giải:

Ý tưởng:

- Ví dụ 1 và 2, các số đều được hình thành từ 2 chữ số, trong đó có 1 số 0, và vì các số tự nhiên đều có nghĩa khi chữ số ở hàng lớn nhất khác 0, do đó lúc đầu queue khởi tạo chỉ bao gồm 01 chữ số khác 0 còn lại.
- Ở ví dụ 3 này, các số vẫn được hình thành từ 2 chữ số, trong đó cả 2 đều khác 0, do đó ngay từ ban đầu phải khởi tạo trong queue có cả 2 chữ số đó. Do đặc điểm của tính chất FIFO nên các số được hình thành theo thứ tự tăng dần. Vì vậy, cần thực hiện một hàm print() để in ra theo đúng yêu cầu đề bài, các thao tác còn lại thực hiện giống như 2 ví dụ trước đó.

Độ phức tạp: O(N) với N là giới hạn trên của M.

Code C++:

```
#include <iostream>
#include <queue>
#include <cstring>
#include <vector>

using namespace std;

using ll = long long;

void print(vector<string> s, int limit)
{
    vector<string> tmp;
    for (string i : s) {
        if (i.length() == limit + 1) break;
        tmp.push_back(i);
    }
    for (vector<string>::reverse_iterator i = tmp.rbegin(); i != tmp.rend();
i++) {
        cout << *(i.base() - 1) << " ";
    }
    cout << endl;
}

vector<string> LuckyNumber (int limit)
{
    queue<string> q;
    vector<string> res;
    q.push("6");
    q.push("8");
    res.push_back("6");
    res.push_back("8");
}
```

```

string num1, num2;
while (true) {
    num1 = q.front() + "6";
    num2 = q.front() + "8";
    res.push_back(num1);
    res.push_back(num2);
    q.push(num1);
    q.push(num2);
    q.pop();
    if (num2.length() == 15) break;
}
return res;
}

int main()
{
    int limit;
    cin >> limit;
    vector<string> res = LuckyNumber(limit);
    print(res, limit);
}

```

Ví dụ 3:

Bài toán tìm số lượng hành động tối thiểu cần thực hiện để chuyển số nguyên a thành số nguyên b .

Có 2 hành động cho phép: nhân với 2 và trừ đi 1.

Ví dụ:

- $a = 2, b = 5$, yêu cầu tương đương với tìm số lượng hành động tối thiểu (bao gồm nhân với 2 và trừ đi 1) để biến đổi số 2 thành số 5.

- Nhận thấy:

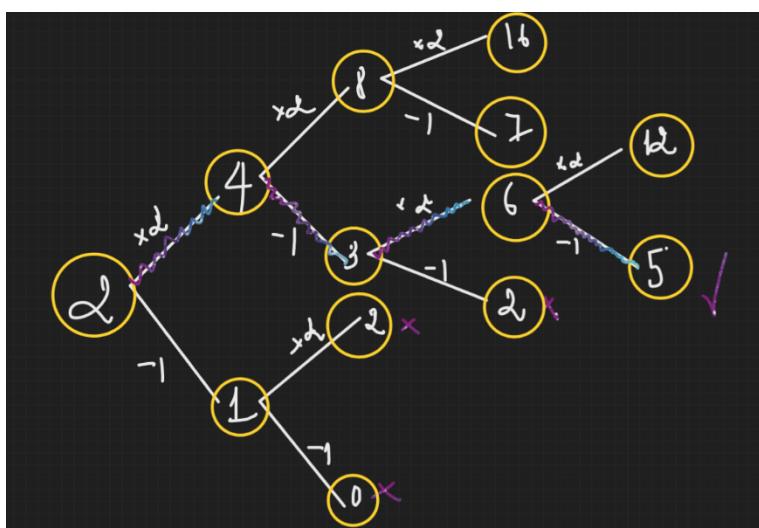
 - $2 * 2 = 4;$
 - $4 - 1 = 3;$
 - $3 * 2 = 6;$
 - $6 - 1 = 5;$

- Do có 4 hành động và số hành động này là tối thiểu (các bước làm đã tối ưu), nên đáp án là 4.

Lời giải:

Ý tưởng:

- Bài toán có thể đưa về dạng sinh số tiếp theo bằng các số trước đó. Đó chính là hơi thở của hướng đi sử dụng cấu trúc dữ liệu hàng đợi queue tương tự như 3 ví dụ trước.
- Ở ví dụ này, ta đưa số cần biến đổi a vào queue, bắt đầu sinh ra các số tiếp theo, số đầu tiên được sinh ra do $a * 2$, số thứ hai được sinh ra do $a - 1$. Kiểm tra lần lượt các số này có bằng b hay không, nếu có thì trả về số hành động đã làm tới thời điểm hiện tại, ngược lại thì thực hiện tiếp công việc này cho đến khi tìm được đáp án.
- Để nắm được số hành động cần làm để đạt được b (tạm gọi là **number**), ta cần kiểm soát đồng thời **number** và giá trị ứng với nó. Chính xác hơn, với mỗi số được sinh ra, ta cần nắm được **number** của riêng nó. Công việc này có thể thực thi bằng cấu trúc dữ liệu pair cho 2 số nguyên. Và do đó queue ở ví dụ này sẽ chứa các tập hợp pair<int, int> thay vì các số nguyên đơn lẻ như 3 ví dụ trước.
- Ngoài ra, trong quá trình thực hiện *đầu dòng* 2, sẽ xảy ra hiện tượng số được sinh ra trùng với một hoặc nhiều các số đã xét trước đó làm cho chương trình thực thi nhiều lần với cùng một dữ liệu. Điều này sẽ làm tốn nhiều bộ nhớ và có thể khiến cho độ phức tạp thời gian tăng đáng kể (một hạn chế khi giải bài toán bằng đệ quy). Để khắc phục điều này, ta cần đưa vào một **SET** để quản lý việc các số chuẩn bị xét có trùng lặp với các số đã xuất hiện trước đó hay không, nếu có thì không xét, ngược lại thì tiếp tục thực thi chương trình như dự tính.
- Mô phỏng:



Độ phức tạp:

Code C++:

```
#include <iostream>
#include <queue>
#include <utility>
#include <set>

using namespace std;

int convertAtoB (int a, int b)
{
    queue<pair<int, int>> q;      // Hàng đợi với mỗi phần tử là 1 pair, 1: giá trị tại chốt, 2: độ dài đường đi tới chốt
    set<int> s;                  // Set kiểm tra chốt chuẩn bị xét đã xét chưa
    q.push({a, 0});              // Đẩy vào queue chốt đầu tiên: giá trị ban đầu là a, đường đi tới chốt là 0
    s.insert(a);                 // Sau khi đã đẩy vào queue, thêm chốt đã xét đó vào set

    while (!q.empty())
    {
        pair<int, int> top = q.front();
        q.pop();
        if (top.first == b) return top.second;
        if (top.first * 2 == b || top.first - 1 == b) return top.second + 1;
        if (s.find(top.first * 2) == s.end() && top.first < b) {
            q.push({top.first * 2, top.second + 1});
            s.insert(top.first * 2);
        }
        if (s.find(top.first - 1) == s.end() && top.first > 1) {
            q.push({top.first - 1, top.second + 1});
            s.insert(top.first - 1);
        }
    }

    int main()
    {
        int a, b;
        cin >> a >> b;
        cout << convertAtoB(a, b);

    }
```

Ví dụ 3:

Bài toán tìm đường đi ngắn nhất trên lưới.

Cho một mảng 2 chiều $a[N][M]$ (với $0 < N, M < 1000$), mô tả một đồ thị lưới, các phần tử trong a được hình thành chỉ từ hai chữ số 0 và 1, hai số 1 nằm cạnh nhau tạo thành một đường có thể đi được, số 0 không được phép đi qua.

Cho vị trí bắt đầu là $A(x, y)$ và vị trí kết thúc là $B(x1, y1)$, hãy tìm đường đi ngắn nhất có thể có được tạo thành từ các chữ số 1 kề nhau trong đồ thị.

Ví dụ:

- Cho dữ liệu vào một file txt, dòng đầu tiên gồm 6 số liên tiếp, bao gồm: số hàng, cột của đồ thị, tọa độ 2 điểm bắt đầu A và kết thúc B. Tiếp theo là đồ thị kích thước [hàng][cột] vừa nhập:

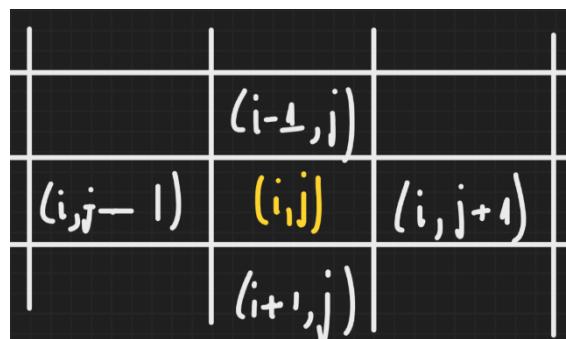
9	1	0	0	0	3	4
1	0	1	1	1	1	0
1	0	1	0	1	1	0
1	1	1	0	1	1	0
0	0	0	0	1	0	0
1	1	1	0	1	1	0
1	0	1	1	1	0	1
1	0	0	0	0	0	1
1	0	1	1	1	0	1
1	1	0	0	0	1	0

- Đầu ra trả về số bước đi (bao gồm số chữ số 1 trừ 1) của đường đi đó, trường hợp này đi qua 12 số 1, tức số bước đi (số bước dịch chuyển) là 11. Đáp án cuối cùng là 11.

Lời giải:

Ý tưởng:

- Duyệt qua tất cả các phần tử, với phần tử ở vị trí (i, j) , xét tất cả 4 vị trí xung quanh:



- Nếu 1 trong 4 phần tử đó là số 1 (đi qua được, có thể duyệt tới được) thì đưa phần tử đó vào queue + tăng số bước đi lên 1 đơn vị, bắt đầu lại quá trình.
- Ngược lại, bỏ qua phần tử đó. Nếu tất cả các phần tử đều không đi được nữa, thì trả về -1 nhằm thông báo rằng không tồn tại đường đi đến điểm đích.

Độ phức tạp:

Code C++:

```
#include <iostream>
#include <queue>
#include <cstring>
#include <utility>
#include <fstream>

using namespace std;

int a[1000][1000];
int d[1000][1000];
int dx[4] = {-1, 0, 0, 1};
int dy[4] = {0, -1, 1, 0};
int hang, cot;

pair<int, int> s, e;

void init(void)
{
    ifstream input;
    input.open("Input.txt");
    input >> hang >> cot >> s.first >> s.second >> e.first >> e.second;
    for (int i = 0; i < hang; i++) {
        for (int j = 0; j < cot; j++) {
            input >> a[i][j];
        }
    }
    memset(d, 0, sizeof(d));
    input.close();
}

void printD (void)
{
    for (int i = 0; i < hang; i++) {
        for (int j = 0; j < cot; j++) {
            cout << d[i][j] << " ";
        }
        cout << endl;
    }
}
```

```

int solve(void)
{
    queue<pair<int, int>> q;
    q.push(s);
    d[s.first][s.second] = 0;

    while (!q.empty())
    {
        pair<int, int> top = q.front(); q.pop();
        int i = top.first;
        int j = top.second;
        for (int k = 0; k < 4; k++) {
            int i1 = i + dx[k];
            int j1 = j + dy[k];
            if (a[i1][j1] && (i1 >= 0) && (i1 < hang) && (j1 >= 0) && (j1 <
cot)) {
                a[i1][j1] = 0;
                if (i1 == e.first && j1 == e.second) {
                    printD();
                    return d[i][j] + 1;
                }
                q.push({i1, j1});
                d[i1][j1] = d[i][j] + 1;
            }
        }
    }
    return -1;
}

int main()
{
    init();
    cout << solve();
}

```

b. Hàng đợi ưu tiên (Priority_Queue):

Nhận xét: Cấu trúc dữ liệu trừu tượng hàng đợi ưu tiên được sử dụng để xử lý các bài toán cần nhiều lần sắp xếp các dữ liệu. Trong khi các thuật toán sắp xếp cần ít nhất $O(N \log N)$ thì Priority_Queue chỉ cần $O(\log N)$ để sắp xếp qua thao tác heapify.

Ví dụ 4:

Bài toán nối dây. Tính chi phí nối dây nhỏ nhất. Chi phí nối 2 sợi dây riêng biệt bằng tổng độ dài 2 sợi dây đó.

Ví dụ:

- $arr[4] = \{4, 2, 3, 6\}$ thì giá trị trả về là 29.
- $arr[6] = \{4, 2, 7, 6, 9\}$ thì giá trị trả về là 62.

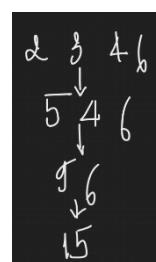
Lời giải:

Ý tưởng:

- Đầu vào là một mảng (vector) chứa N sợi dây, mỗi phần tử mang giá trị của độ dài sợi dây đó. Mảng này cần được sắp xếp theo thứ tự tăng dần.
- Duyệt qua tất cả các phần tử trong mảng, mỗi lần duyệt lấy ra 2 phần tử nhỏ nhất (bỏ ra khỏi mảng), tính tổng **SUM** rồi đưa trở lại mảng, sắp xếp mảng lại theo thứ tự tăng dần.
- Tuy nhiên với các thuật toán đã học, mỗi lần duyệt và sắp xếp các phần tử ta cần ít nhất $O(N \log N)$ thông qua các thuật toán sắp xếp, do nhu cầu sắp xếp lớn nên phương pháp dùng mảng để giải quyết là khả thi nhưng không tối ưu.
- Hàng đợi ưu tiên với MIN HEAP (phần tử nhỏ nhất có độ ưu tiên cao nhất) có thể giải quyết được vấn đề này do các thao tác chèn 1 phần tử vào hàng đợi có bản chất là thêm 1 phần tử $O(1)$ vào mảng, và sắp xếp bằng thao tác Heapify ($O(\log N)$). Do đó độ phức tạp giảm còn $O(\log N)$.

Độ phức tạp: $O(N.A)$ với N là số phần tử ban đầu (số sợi dây) và $A = \sum_{i=1}^N \log(i)$. i là

số phần tử trong mảng hiện tại, mô phỏng như sau:



Code C++:

```
int maxCost (vector<int> drop)
{
    priority_queue<int, vector<int>, greater<int>> pq;
    for (int i : drop) {
        pq.push(i);
    }
    int res = 0;
    while (pq.size() > 1) {
        int f = pq.top(); pq.pop();
        int s = pq.top(); pq.pop();
        res += f + s;
        pq.push(f + s);
    }
    return res;
}
```

Ví dụ 5:

Bài toán tìm giá trị nhỏ nhất của xâu.

Cho một xâu s bao gồm các chữ cái in hoa, giá trị của xâu được tính bằng tổng các bình phương số lần xuất hiện của từng xâu. Cho một số tự nhiên K , có thể bỏ tối đa K kí tự của xâu s . Tìm giá trị nhỏ nhất của xâu.

Ví dụ:

- $s = "AAABBCD"$, $K = 1$ (tức có thể bỏ tối đa 1 kí tự trong xâu s);
- Giá trị ban đầu của xâu: $3^2 + 2^2 + 1^2 + 1^2 = 15$.
- Giá trị nhỏ nhất của xâu: $2^2 + 2^2 + 1^2 + 1^2 = 10$.
- Đáp án cuối cùng là 10;

Lời giải:

Ý tưởng:

- Giá trị sẽ không thể nhỏ nhất nếu không bỏ đi K kí tự, tuy nhiên sau khi bỏ K kí tự bất kí, giá trị của xâu chưa chắc nhỏ nhất. Muốn đạt giá trị nhỏ nhất, phải

bỏ đi 1 kí tự có tần suất xuất hiện nhiều nhất, vì khi đó sẽ giảm được cơ số của 1 bình phương.

- Như vậy, công việc cần làm là tìm ra tần suất xuất hiện của tất cả các kí tự riêng biệt trong xâu s, lưu vào một mảng. Sau đó tìm phần tử lớn nhất trong mảng, bắt đầu giảm K kí tự, nếu kí tự hiện tại đã giảm hết (tức không còn kí tự đó trong xâu) thì sẽ giảm kí tự có tần suất bé hơn liền kề đó.
- Để giảm thiểu công sức thực thi code, ta có thể sắp xếp mảng tần suất theo thứ tự giảm dần, và bắt đầu giảm K kí tự bắt đầu từ phần tử đầu tiên, lần lượt đến khi K kí tự đã được xóa bỏ. Tuy nhiên các thuật toán sắp xếp tốn ít nhất $O(N \log N)$, do đó cấu trúc hàng đợi ưu tiên được đưa vào khi các thao tác đẩy vào 1 phần tử chỉ tốn $O(\log N)$.
- Associative Container map sẽ được sử dụng để tính tần suất các kí tự, keys là các kí tự riêng biệt, mapped values là số nguyên biểu diễn tần suất.

Độ phức tạp: $O(N)$ với N là độ dài xâu s.

Code C++:

```
#include <iostream>
#include <queue>
#include <map>
#include <set>

using namespace std;

long int maxString (string s, int K)
{
    if (K >= s.length()) return 0;
    priority_queue<int, vector<int>, less<int>> pq;
    long int res = 0;
    map<char, int> mp;

    for (char i : s) {
        mp[i]++;
    }
    for (pair<char, int> i : mp) {
        pq.push(i.second);
    }
    while (K--) {
        int top = pq.top(); pq.pop();
        top--;
        res++;
    }
}
```

```
        pq.push(top);
    }
    while (!pq.empty()) {
        int top = pq.top(); pq.pop();
        res += top * top;
    }
    return res;
}

int main()
{
    string s; cin >> s;
    int K; cin >> K;
    cout << maxString(s, K);
}
```

III. CÂU TRÚC DỮ LIỆU DANH SÁCH LIÊN KẾT (LINKED LIST):

1. Tổng quan:

Là một cấu trúc dữ liệu trừu tượng. Danh sách liên kết không phải là một kiểu dữ liệu trong ngôn ngữ C++ hay các ngôn ngữ lập trình khác. Thay vào đó, danh sách liên kết là một cấu trúc dữ liệu được xây dựng từ các phần tử riêng lẻ gọi là "node" và các node này kết nối với nhau thông qua các con trỏ.

2. Các dạng toán sử dụng danh sách liên kết:

3. Cách cài đặt danh sách liên kết trong ngôn ngữ lập trình C++:

a. Danh sách liên kết đơn (Singly Linked List):

```
#include <iostream>

using namespace std;

struct node
{
    int data;
    node *next;
};

node* createNode (int a)
{
    node *newNode = new node();
    newNode -> data = a;
    newNode -> next = nullptr;
    return newNode;
}

void duyet (node *head)
{
    node *tmpNode = head;
    while (tmpNode != nullptr)
    {
        cout << tmpNode -> data << " ";
        tmpNode = tmpNode -> next;
    }
}

int count (node *head)
{
    int count = 0;
```

```

node *tmpNode = head;
while (tmpNode != nullptr)
{
    count++;
    tmpNode = tmpNode -> next;
}
return count;
}

void pushFront (node *&head, int a)
{
    node *newNode = createNode(a);
    node *tmpNode = head;
    if (head == nullptr) {
        head = newNode;
    }
    else {
        newNode -> next = tmpNode;
        head = newNode;
    }
}

void pushIn (node *&head, int a, int index)
{
    node *tmpNode = head;
    node *newNode = createNode(a);
    int size = count(head);
    if (head == nullptr) {
        head = newNode;
    }
    else if (index <= 0 || index > size) {
        return;
    }
    else if (index == 1) {
        pushFront (head, a);
    }
    else {
        for (int i = 1; i <= index - 2; i++) {
            tmpNode = tmpNode -> next;
        }
        newNode -> next = tmpNode -> next;
        tmpNode -> next = newNode;
    }
}

void pushBack (node *&head, int a)
{
    node *newNode = createNode(a);

```

```

node *tmpNode = head;
while (tmpNode -> next != nullptr) {
    tmpNode = tmpNode -> next;
}
tmpNode -> next = newNode;
}

void popFront (node *&head)
{
    node *tmpNode = head;
    if (head == nullptr) {
        cout << "There is no node to remove from Front ward";
    }
    else {
        head = head -> next;
        delete tmpNode;
    }
}

void popIn (node *&head, int index)
{
    node *tmpNode = head;
    int size = count (head);
    if (head == nullptr) {
        cout << "There is no node to remove from In";
    }
    else if (index <= 0 || index > size) {
        cout << "Invalid index.";
    }
    else {
        for (int i = 1; i <= index - 2; i++) {
            tmpNode = tmpNode -> next;
        }
        node *removedNode = tmpNode -> next;
        tmpNode -> next = tmpNode -> next -> next;
        delete removedNode;
    }
}

void popBack (node *&head)
{
    node *tmpNode = head;
    if (head == nullptr) {
        cout << "There is no node to remove from Back";
    }
    else if (head -> next == nullptr) {
        head = nullptr;
        delete tmpNode;
    }
}

```

```

    }
    else {
        while (tmpNode -> next -> next != nullptr) {
            tmpNode = tmpNode -> next;
        }
        node *lastNode = tmpNode -> next;
        tmpNode -> next = nullptr;
        delete lastNode;
    }
}

void reverseList (node *&head)
{
    if (head == nullptr || head -> next == nullptr) return;
    node *preNode = nullptr;
    node *curNode = head;
    node *nextNode;
    while (curNode != nullptr) {
        nextNode = curNode -> next;
        curNode -> next = preNode;
        preNode = curNode;
        curNode = nextNode;
    }
    head = preNode;
}

int main()
{
    node *head = nullptr;
    for (int i = 1; i <= 5; i++) {
        pushFront (head, i);
    }
    duyet(head);
    cout << endl << endl;

    pushIn(head, 432, 4);
    duyet(head);
    cout << endl << endl;

    for (int i = 100; i <= 105; i++) {
        pushBack (head, i);
    }
    duyet(head);
    cout << endl << endl;

    popFront (head);
    duyet(head);
    cout << endl;
}

```

```

cout << "head: " << head -> data << endl << endl;

popBack(head);
duyet(head);
cout << endl;
cout << "head: " << head -> data << endl << endl;

popIn(head, 3);
duyet(head);
cout << endl;
cout << "head: " << head -> data << endl << endl;
}

```

b. Danh sách liên kết kép (Doubly Linked List):

```

#include <iostream>

using namespace std;

#define _ DLL.
#define INN printNext()
#define INP printPrev()

class DoublyLinkedList
{
    struct node {
        int data;
        node *prev;
        node *next;
    };
    static node* createNode (int a) {
        node *newNode = new node();
        newNode -> data = a;
        newNode -> prev = nullptr;
        newNode -> next = nullptr;
        return newNode;
    }
    node *DSLK = nullptr;
    node *&head = DSLK;
public:
    DoublyLinkedList();
    void printNext () const;
    void printPrev () const;
    int count () const;
    void pushFront (int a);
    void pushIn (int a, int);
}

```

```

    void pushBack (int a);
    void popFront ();
    void popIn (int);
    void popBack ();
    ~DoublyLinkedList();
};

DoublyLinkedList::DoublyLinkedList()
{
    cout << "Constructor creates a new doubly linked list with initializer
head = nullptr." << endl;
}

void DoublyLinkedList::printNext () const
{
    for (node *i = head; i != nullptr; i = i->next) {
        cout << i -> data << " ";
    }
    cout << endl;
}

void DoublyLinkedList::printPrev () const
{
    if (head == nullptr) {
        return;
    }
    node *tmpNode = head;
    for (tmpNode; tmpNode -> next != nullptr; tmpNode = tmpNode -> next) {}
    for (tmpNode; tmpNode != nullptr; tmpNode = tmpNode->prev) {
        cout << tmpNode -> data << " ";
    }
    cout << endl;
}

int DoublyLinkedList::count () const
{
    int count = 0;
    for (node *i = head; i != nullptr; i = i -> next) {
        count++;
    }
    return count;
}

void DoublyLinkedList::pushFront (int a)
{
    node *newNode = DoublyLinkedList::createNode(a);
    if (head == nullptr) {
        head = newNode;
    }
}

```

```

    }
    else {
        newNode -> next = head;
        head -> prev = newNode;
        head = newNode;
    }
}

void DoublyLinkedList::pushIn (int a, int index)
{
    if (index < 1 || index > count()) {
        return;
    }
    else if (index == 1) {
        return pushFront(a);
    }
    node *tmpNode = head;
    node *newNode = DoublyLinkedList::createNode(a);
    if (head == nullptr) {
        head = newNode;
    }
    else {
        for (int i = 1; i <= index - 1; i++) {
            tmpNode = tmpNode -> next;
        }
        newNode -> next = tmpNode;
        tmpNode -> prev -> next = newNode;
        newNode -> prev = tmpNode -> prev;
        tmpNode -> prev = newNode;
    }
}

void DoublyLinkedList::pushBack (int a)
{
    node *tmpNode = head;
    node *newNode = DoublyLinkedList::createNode(a);
    if (head == nullptr) {
        head = newNode;
    }
    else {
        while (tmpNode -> next != nullptr) {
            tmpNode = tmpNode -> next;
        }
        tmpNode -> next = newNode;
        newNode -> prev = tmpNode;
    }
}

```

```
void DoublyLinkedList::popFront ()
{
    node *tmpNode = head;
    if (head == nullptr) {
        return;
    }
    else {
        head = head -> next;
        head -> prev = nullptr;
        delete tmpNode;
    }
}

void DoublyLinkedList::popBack ()
{
    node *tmpNode = head;
    if (head == nullptr) {
        return;
    }
    else if (head -> next == nullptr) {
        head = nullptr;
    }
    else {
        while (tmpNode -> next -> next != nullptr) {
            tmpNode = tmpNode -> next;
        }
        node *lastNode = tmpNode -> next;
        tmpNode -> next = nullptr;
        delete lastNode;
    }
}

void DoublyLinkedList::popIn (int index)
{
    if (index < 1 || index > count()) {
        return;
    }
    else if (index == 1) {
        popFront();
    }
    node *tmpNode = head;
    if (head == nullptr) {
        return;
    }
    else if (head -> next == nullptr) {
        head = nullptr;
    }
    else if (head -> next -> next == nullptr) {
```

```

        if (index == 1) {
            popFront();
        }
        else {
            popBack();
        }
    }
else {
    for (int i = 1; i <= index - 1; i++) {
        tmpNode = tmpNode -> next;
    }
    tmpNode -> prev -> next = tmpNode -> next;
    tmpNode -> next -> prev = tmpNode -> prev;
    delete tmpNode;
}
}

DoublyLinkedList::~DoublyLinkedList()
{
    while (head != nullptr) {
        node *tmpNode = head;
        head = head -> next;
        delete tmpNode;
    }
    if (head == nullptr) {
        cout << "Removed all." << endl;
    }
    delete DSLK;
    delete head;
}

int main()
{
    DoublyLinkedList DLL;
    _ pushFront(3);
    _ pushFront(2);
    _ pushFront(1);
    for (int i = 8; i >= 5; i--) {
        _ pushBack(i);
    }
    _ pushIn(4, 2);
    _ INN;
    _ INP;

    _ popFront(); _ INN;
    _ popIn(2); _ INN;
    _ popBack(); _ INN;
    _ INP;
}

```

```
}
```

Cách tối ưu hơn:

```
#include <iostream>

using namespace std;

struct node
{
    int data;
    node *prev;
    node *next;
    node() : data(0), prev(nullptr), next(nullptr) {}
    node (int a) : data(a), prev(nullptr), next(nullptr){}
};

class DoublyLinkedList
{
    node *head;
    int count;
public:
    DoublyLinkedList();
    ~DoublyLinkedList();
    void printPrev() const;
    void printNext() const;
    void pushFront(int a);
    void pushIn(int a, int index);
    void pushBack(int a);
    void popFront();
    void popIn(int index);
    void popBack();
};

DoublyLinkedList::DoublyLinkedList()
{
    head = nullptr;
    count = 0;
}

DoublyLinkedList::~DoublyLinkedList()
{
    while (head != nullptr) {
        node *tmpNode = head;
        head = head -> next;
        delete tmpNode;
        --count;
    }
}
```

```

head = nullptr;
if (count == 0) {
    cout << "Entire list is removed." << endl;
}
}

void DoublyLinkedList::printPrev() const
{
    node *tmpNode = head;
    if (this->count == 0) {
        return;
    }
    else {
        for (tmpNode; tmpNode -> next != nullptr; tmpNode = tmpNode -> next)
    }

        cout << count << " : ";
        for (int i = 0; i < this->count; i++) {
            cout << tmpNode -> data << " ";
            tmpNode = tmpNode -> prev;
        }
        cout << endl;
    }
}

void DoublyLinkedList::printNext() const
{
    node *tmpNode = head;
    if (this->count == 0) {
        return;
    }
    else {
        cout << count << " : ";
        while (tmpNode != nullptr) {
            cout << tmpNode->data << " ";
            tmpNode = tmpNode -> next;
        }
        cout << endl;
    }
}

void DoublyLinkedList::pushFront(int a)
{
    node *newNode = new node(a);
    if (this->count == 0) {
        this->head = newNode;
    }
    else {

```

```

        newNode -> next = head;
        head -> prev = newNode;
        head = newNode;
    }
    ++count;
}

void DoublyLinkedList::pushIn(int a, int index)
{
    if (index < 0 || index >= count) {
        return;
    }
    else if (index == 0) {
        return pushFront(a);
    }
    else {
        node *newNode = new node(a);
        node *tmpNode = head;
        for (int i = 0; i < index - 1; i++) {
            tmpNode = tmpNode -> next;
        }
        newNode -> next = tmpNode -> next;
        tmpNode -> next -> prev = newNode;
        newNode -> prev = tmpNode;
        tmpNode -> next = newNode;
        ++count;
    }
}

void DoublyLinkedList::pushBack(int a)
{
    node *newNode = new node(a);
    if (this->count == 0) {
        head = newNode;
    }
    else {
        node *tmpNode = head;
        for (int i = 0; i < count - 1; ++i) {
            tmpNode = tmpNode -> next;
        }
        tmpNode -> next = newNode;
        newNode -> prev = tmpNode;
    }
    ++count;
}

void DoublyLinkedList::popFront()
{

```

```

    if (count == 0) {
        cout << "No element to popFront" << endl;
    }
    else {
        node *deleteNode = head;
        head = head -> next;
        head -> prev = nullptr;
        delete deleteNode;
        --count;
    }
}

void DoublyLinkedList::popIn(int index)
{
    if (index < 0 || index >= count) {
        cout << "Invalid index" << endl;
    }
    else if (index == 0) {
        return popFront();
    }
    else {
        node *tmpNode = head;
        for (int i = 0; i < index; i++) {
            tmpNode = tmpNode -> next;
        }
        tmpNode -> prev -> next = tmpNode -> next;
        if (index != count - 1) {
            tmpNode -> next -> prev = tmpNode -> prev;
        }
        delete tmpNode;
        --count;
    }
}

void DoublyLinkedList::popBack()
{
    if (count == 0) {
        cout << "There is no element to popBack" << endl;
    }
    else if (count == 1) {
        delete head;
        head = nullptr;
        --count;
    }
    else {
        node *tmpNode = head;
        for (int i = 0; i < count - 1; i++) {
            tmpNode = tmpNode -> next;
        }
        tmpNode -> prev -> next = nullptr;
        delete tmpNode;
        --count;
    }
}

```

```

        }
        tmpNode -> prev -> next = nullptr;
        delete tmpNode;
        --count;
    }
}

int main()
{
    DoublyLinkedList DLL;
    DLL.pushFront(4);
    for (int i = 0; i < 6; i++) {
        DLL.pushBack(i);
    }
    DLL.pushIn(13, 4);
    DLL.printNext();
    //DLL.printPrev();

    DLL.popFront(); DLL.printNext();
    DLL.popIn(4); DLL.printNext();
    DLL.popBack(); DLL.printNext();
    DLL.printPrev();
}

```

c. Danh sách liên kết kép vòng (Doubly Circular Linked List):

```

#include <iostream>

using namespace std;

struct node
{
    int data;
    node *prev;
    node *next;
    node() : data(0), prev(nullptr), next(nullptr) {}
    node (int a) : data(a), prev(nullptr), next(nullptr){}
};

class DoublyCircularLinkedList
{
public:
    node *head;

```

```
    int count;
public:
    DoublyCircularLinkedList();
    ~DoublyCircularLinkedList();
    void print(int index, int num) const;
    void push(int index, int a);
    void pop(int index);
};

DoublyCircularLinkedList::DoublyCircularLinkedList()
{
    head = nullptr;
    count = 0;
}

DoublyCircularLinkedList::~DoublyCircularLinkedList()
{
    for (int i = 0; i < count; i++) {
        head -> prev = nullptr;
        head = head -> next;
    }
    node *tmpNode = head -> next;

    head -> next = nullptr;
    while (tmpNode != nullptr) {
        node *deleteNode = tmpNode;
        tmpNode = tmpNode -> next;
        delete deleteNode;
        --count;
    }
    if (count == 0) {
        cout << "The entire list was removed." << endl;
    }
}

void DoublyCircularLinkedList::print(int index, int num) const
{
    node *tmpNode = head;
    if (count == 0) {
        cout << "Empty list." << endl;
        return;
    }
    else {
        if (index >= count) {
            index %= count;
        }
        for (int i = 0; i < index; i++) {
            tmpNode = tmpNode -> next;
        }
        cout << tmpNode->val << endl;
    }
}
```

```

        }
        for (int i = 0; i < num; i++) {
            cout << tmpNode -> data << " ";
            tmpNode = tmpNode -> next;
        }
        cout << endl;
    }
}

void DoublyCircularLinkedList::push(int index, int a)
{
    node *newNode = new node(a);
    if (count == 0) {
        head = newNode;
        head -> next = head;
        head -> prev = head;
        ++count;
    }
    else {
        node *tmpNode = head;
        if (index == 0) {
            head = newNode;
        }

        if (index >= count) {
            index %= count;
        }
        if (index < count/2) {
            for (int i = 0; i < index; ++i) {
                tmpNode = tmpNode -> next;
            }
        }
        else {
            for (int i = count; i > index; --i) {
                tmpNode = tmpNode -> prev;
            }
        }
        tmpNode -> prev -> next = newNode;
        newNode -> prev = tmpNode -> prev;
        newNode -> next = tmpNode;
        tmpNode -> prev = newNode;
        ++count;
    }
}

void DoublyCircularLinkedList::pop(int index)
{
    if (count == 0) {

```

```

        cout << "There is no node to pop." << endl;
    }
else {
    node *tmpNode = head;
    if (index == 0) {
        head = head -> next;
    }

    if (index >= count) {
        index %= count;
    }
    if (index < count/2) {
        for (int i = 0; i < index; i++) {
            tmpNode = tmpNode -> next;
        }
    }
    else {
        for (int i = count; i > index; --i) {
            tmpNode = tmpNode -> prev;
        }
    }
    tmpNode -> prev -> next = tmpNode -> next;
    tmpNode -> next -> prev = tmpNode -> prev;
    delete tmpNode;
    --count;
}

}

int main()
{
    DoublyCircularLinkedList DCLL;
    DCLL.push(4, 12);
    for (int i = 0; i < 7; i++) {
        DCLL.push(i, i);
    }
    DCLL.print(0, DCLL.count);
    cout << endl;

    DCLL.pop(DCLL.count - 1);
    for (int i = 0; i < 3; i++) {
        DCLL.pop(i);
    }
    DCLL.print(0, DCLL.count);
}

```

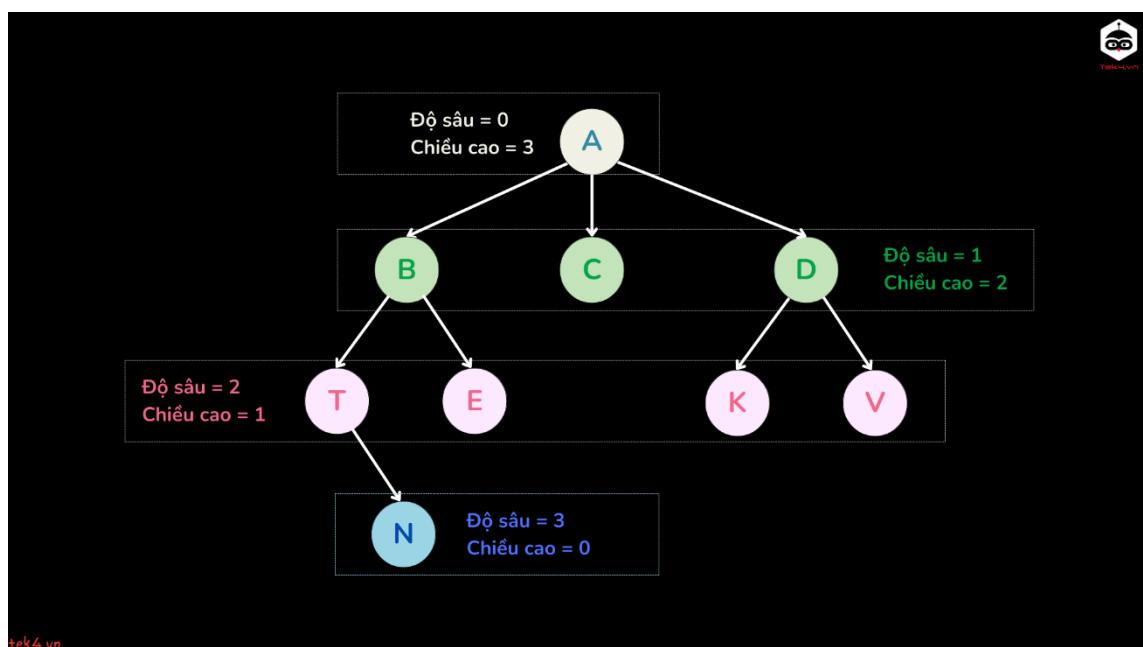
IV. CÂU TRÚC DỮ LIỆU CÂY (TREE):

1. Tổng quan:

Là cấu trúc dữ liệu trừu tượng.

Cây là một dạng đơn đồ thị (simple graph):

- Có hướng (directed);
- Liên thông (connected);
- Không có chu trình (no circuits);
- Đỉnh có bậc vào (in – degree) bằng 0: gốc (root);
- Đỉnh có bậc vào (in – degree) bằng 1: tất cả các đỉnh còn lại, gồm đỉnh trong (internal vertices) và lá (leaves).
- Mức (level) **của 1 đỉnh**: độ dài đường đi từ gốc đến đỉnh đang xét (số cạnh hiện hữu), mặc định gốc có level bằng 0;
- Chiều cao (height) **của 1 cây**: mức lớn nhất (largest level) – độ dài đường đi dài nhất trong cây.
- Với mỗi nút (đỉnh):
 - + Độ sâu: là 1 cách gọi khác của mức (level);
 - + Chiều cao: độ dài đường đi từ nút đang xét đến nút lá;
 - + Như vậy: *tổng độ sâu và chiều cao của một tập hợp các nút có cùng level luôn bằng height của cả cây đang xét.*



2. Các dạng toán sử dụng cấu trúc dữ liệu cây:

- Các bài toán xử lý mảng (tìm kiếm, sắp xếp, ...) có thể chuyển về cấu trúc cây để giải quyết, vì hầu hết các thao tác riêng rẽ với cây (truy cập các nút, xóa, thêm, ...) đều có độ phức tạp từ $O(\log N)$ trở xuống. <Với mảng thường là $O(N)$ >. Diễn hình của dạng này là: Binary Search (tìm kiếm nhị phân), Heap Sort (thuật toán sắp xếp vun đống), ...
- Các bài toán liên quan đến duyệt và in infix, prefix, postfix, ... (có thể sử dụng stack để thao tác).
- Xây dựng hàng đợi ưu tiên (priority queue) bằng cấu trúc dữ liệu được khai triển HEAP, thông qua thao tác HEAPIFY.

3. Các ví dụ điển hình:

a. Cấu trúc dữ liệu cây nhị phân (Binary Tree):

- *Cây nhị phân*: là cây mà mỗi nút cha có tối đa 2 cây con: cây con trái và cây con phải.
- *Tính chất*:
 - + Số node nằm ở level i (mức i) $\leq 2^i$;
 - + Số node lá $\leq 2^h$ (h là chiều cao của cây);
 - + Cây nhị phân có chiều cao là h ($h \geq 0$) sẽ có tối đa: $2^{h+1} - 1$ node;
 - + Chiều cao của cây: $h \geq \log_2(N)$ (N là số node trong cây);
- Chứng minh dấu + thứ 3:
 - + Giả sử một cây nhị phân chiều cao $h = 3$, khi đó số node tối đa là 15;
 - + Cách tính:
 - Số node tối đa ở mức 0: $2^0 = 1$;
 - Số node tối đa ở mức 1: $2^1 = 2$;
 - Số node tối đa ở mức 2: $2^2 = 4$;
 - Số node tối đa ở mức 3: $2^3 = 8$;

- Như vậy, số node tối đa của cây nhị phân chiều cao $h = 3$ là

$$\begin{aligned}
 & 2^0 + 2^1 + 2^2 + 2^3 = 15 \\
 \Leftrightarrow & 2^0 + 2^1 + 2^2 + 2^3 + (2^1 - 2^1) = 15 \\
 \Leftrightarrow & \underbrace{2^1 + 2^1}_{2^2} + 2^2 + 2^3 + \underbrace{(2^0 - 2^1)}_{-1} = 15 \\
 \Leftrightarrow & 2 \times 2^2 + 2^3 - 1 = 15 \\
 \Leftrightarrow & 2^3 + 2^3 - 1 = 15 \Leftrightarrow 2 \times 2^3 - 1 = 15 \Leftrightarrow 2^{3+1} - 1 = 15
 \end{aligned}$$

+ Tổng quát bài toán, ta được công thức như trên.

- Các kiểu cây nhị phân:

- + Cây nhị phân đầy đủ (nghiêm ngặt) (full binary tree / strict binary tree):

- Các node lá không nhất thiết phải ở cùng 1 level.
- Mọi node: có đủ 2 con hoặc không có con nào cả.

- + Cây nhị phân hoàn chỉnh (complete binary tree):

- Tất cả các node lá đều ở cùng 1 level, cùng 1 độ sâu, cùng 1 chiều cao bằng 0.
- Riêng các node lá cuối cùng phải được điền vào bên trái nhất có thể.
- Cây nhị phân hoàn chỉnh là 1 cây nhị phân lí tưởng đến level $h - 1$. Ở level h , các nút được xếp từ trái sang phải.
- Khi biểu diễn bằng mảng, các phần tử được xếp sát nhau, liên tục, không ngắt quãng.

- + Cây nhị phân lí tưởng (perfect binary tree):

- Tất cả các node (trừ node lá) đều filled;
- Số node đúng bằng $2^{h+1} - 1$ (h là chiều cao của cây).

- + Cây nhị phân thoái hóa:

- + Cây nhị phân bị lệch:

- + Cây nhị phân cân bằng:

- Mô phỏng cây nhị phân bằng C++:

```

struct node
{
    int data;
    node *left;
    node *right;
}

```

```

};

node* createNode (int a)
{
    node *newNode = new node();
    newNode -> data = a;
    newNode -> left = nullptr;
    newNode -> right = nullptr;
    return newNode;
}

```

- **Duyệt cây nhị phân:**

- + [Depth First Traversal](#) (Duyệt theo chiều sâu):
 - Pre – order: Node – Left – Right (NLR);
 - In – order: Left – Node – Right (LNR);
 - Post – order: Left – Node – Right (LNR);
- + [Breadth First Traversal](#) – Level Order Traversal (Duyệt theo chiều rộng);
- + Độ phức tạp thao tác duyệt: O(N) với N là số node trong cây cho DFS và với BFS là O(N.k) với N là số node trong cây và k số lượng node trong cùng level đang xét (k là số node trên một hàng).

✓ **Với DFS:**

Sử dụng đệ quy:

```

void Print_PreOrder (node *root)
{
    if (root == nullptr) {
        return;
    }
    cout << root -> data << " ";
    Print_PreOrder(root -> left);
    Print_PreOrder(root -> right);
}

void Print_InOrder (node *root)
{
    if (root == nullptr) {
        return;
    }
    Print_InOrder(root -> left);
    cout << root -> data << " ";

```

```

        Print_InOrder(root -> right);
    }

void Print_PostOrder (node *root)
{
    if (root == nullptr) {
        return;
    }
    Print_PostOrder(root -> left);
    Print_PostOrder(root -> right);
    cout << root -> data << " ";
}

```

Sử dụng vòng lặp:

```

void Print_PreOrder (node *root)
{
    stack<node*> st;
    st.push(root);
    while (!st.empty())
    {
        node *top = st.top(); st.pop();
        cout << top -> data << " ";
        if (top -> right != nullptr) {
            st.push(top -> right);
        }
        if (top -> left != nullptr) {
            st.push(top -> left);
        }
    }
}

```

```

void print_In (node *root)
{
    stack<node*> st;
    bool done = false;
    while (!done)
    {
        if (root != nullptr) {
            st.push(root);
            root = root -> left;
        }
        else {
            if (st.empty()) {
                done = true;
                return;
            }

```

```

        root = st.top(); st.pop();
        cout << root -> data << " ";
        root = root -> right;
    }
}
}

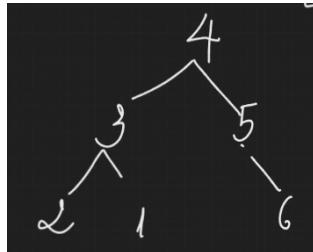
```

```

void Print_PostOrder (node *root)
{
    if (root == nullptr) {
        return;
    }
    stack<node*> st;
    st.push(root);
    node *prev = nullptr;
    while (!st.empty())
    {
        node *curr = st.top();
        if (prev == nullptr || prev -> left == curr || prev -> right == curr)
        {
            if (curr -> left != nullptr) {
                st.push(curr -> left);
            }
            else if (curr -> left == prev && curr -> right != nullptr) {
                st.push(curr -> right);
            }
        }
        else if (curr -> left == prev && curr -> right != nullptr) {
            st.push(curr -> right);
        }
        else {
            cout << curr -> data << " ";
            st.pop();
        }
        prev = curr;
    }
}

```

Lưu ý: Với cách duyệt hậu tố này, ta chỉ áp dụng được cho cây nhị phân hoàn chỉnh (complete binary tree), với những cây nhị phân không hoàn chỉnh, kết quả sẽ không chính xác. Ví dụ:



Thay vì đáp án chính xác là: 2 1 3 6 5 4, output sẽ chỉ là: 2 1 3 5 4, chuỗi hành động sai trong code bắt đầu từ khi duyệt chuyển tiếp từ cây con chính trái sang cây con chính phải.

Do đó ta thay bằng code hiệu quả và tối hơn bằng cách dùng 2 ngăn xếp stack:

```
void Print_Post (node *root)
{
    stack<node*> st;
    stack<int> res;
    st.push(root);
    while(!st.empty())
    {
        node *top = st.top(); st.pop();
        res.push(top -> data);
        if (top -> left != nullptr) {
            st.push(top -> left);
        }
        if (top -> right != nullptr) {
            st.push(top -> right);
        }
    }
    while (!res.empty())
    {
        cout << res.top() << " ";
        res.pop();
    }
}
```

- Thêm node – tạo cây:

```
void pushLeft (node *root, int a)
{
    node *newNode = createNode(a);
    root -> left = newNode;
}

void pushRight (node *root, int a)
{
    node *newNode = createNode(a);
    root -> right = newNode;
}
```

✓ Với BFS:

Cách 1:

```
void BFS (node *root)
{
    queue<node*> qu;
    qu.push(root);
    while (!qu.empty()) {
        node *top = qu.front(); qu.pop();
        cout << top -> data << " ";
        if (top -> left) {
            qu.push(top -> left);
        }
        if (top -> right) {
            qu.push(top -> right);
        }
    }
}
```

Cách 2:

Cách này tuy dài dòng hơn nhưng bù lại nó cung cấp một tiền đề vô cùng hữu ích trong các thao tác với duyệt BFS, vì ta đã cố định mỗi vòng lặp lớn (while loop) sẽ duyệt toàn bộ một hàng (các node cùng level).

```
void Print_BFS (node *root)
{
    if (root == nullptr) {
        cout << "Null tree" << endl;
        return;
    }
    queue<node*> qu;
    qu.push(root);
    while (!qu.empty()) {
        int N = qu.size();
        for (int i = 0; i < N; i++) {
            node *top = qu.front(); qu.pop();
            cout << top -> data << " ";
            if (top -> left != nullptr) {
                qu.push(top -> left);
            }
            if (top -> right != nullptr) {
                qu.push(top -> right);
            }
            top = nullptr;
        }
    }
}
```

}

Lưu ý: Nếu ta sử dụng lệnh `delete top; top = nullptr`, thì toàn bộ cây sẽ bị xóa khỏi bộ nhớ. Điều này không ảnh hưởng đến việc duyệt cây (vì vốn dĩ ta đã in ra giá trị node đó trước khi xóa), tuy nhiên hành động `delete top` dẫn đến việc không thể thực hiện bất kỳ hành động nào với cây tiếp sau duyệt vì cây đã bị khai tử.

Nguyên nhân do câu lệnh `delete top` có nhiệm vụ giải phóng vùng nhớ do con trỏ `top` đang trỏ tới trong bộ nhớ heap (giải phóng vùng nhớ được cấp phát động bởi toán tử `new`), vì `top` đang là một bản sao của `qu.front()` (node đầu tiên trong hàng đợi), nên `top` cũng đang trỏ đến node đang xét trong heap. Sau khi delete, vùng nhớ được giải phóng, tức toàn bộ node trong heap được giải phóng, node chính thức bị xóa. Đó là lí do vì sao toàn bộ cây bị mất.

Việc cho `top = nullptr` có thể có hoặc không, sự tồn tại của câu lệnh này chỉ nhằm chỉ định rằng `top` là một con trỏ null, không gây ảnh hưởng đến cây. Vì câu lệnh `node *top = qu.front();` bản chất là tạo ra một bản sao của `qu.front()`, thay đổi bản sao (`top`) không đồng nghĩa với thay đổi bản chính (`qu.front()`). Khác hoàn toàn so với trường hợp `delete`.

❖ **Ứng dụng DFS (depth first search) vào cây nhị phân:**

Ví dụ 1:

Tìm đường đi dài nhất từ node gốc (root) đến node lá của một cây nhị phân.

Lời giải:

Ý tưởng:

- **Cách 1:**

- + Đếm từ trên gốc xuống các lá theo quy tắc đệ quy;
- + Ứng dụng DFS như cách duyệt cây nhị phân, thay vì cout ra giá trị các node, ta đếm giá trị đường đi tạm thời;
- + Đề quy với biến count, mỗi lần đệ quy, count tăng 1 đơn vị.
- + Lưu ý: thuật toán sẽ chỉ kết thúc khi gặp node là con trỏ null, do vậy biến count sẽ lớn hơn 1 đơn vị so với thực tế mong muốn, như vậy ta cần duyệt count bắt đầu từ -1, vẫn để sẽ được giải quyết.

- **Cách 2:**

- + Đếm từ lá ngược lên gốc theo quy tắc đệ quy;
- + Khi đó các node nullptr trả về giá trị 0, sau đó cộng 1 vào node cha;
- + Nhìn chung, mỗi lần duyệt node, tại node đó trả về 1 giá trị, giá trị này chính là số lượng node trong 1 nhánh cây con trái hoặc phải của node đó. Như vậy, giá trị này được tính bằng *max (số node trong cây con trái, số node trong cây con phải)* + 1;
- + Thứ tự đệ quy tương tự như duyệt cây nhị phân bằng đệ quy với DFS.

- **Cách 3:**

- + Sử dụng ngăn xếp và vòng lặp thay cho đệ quy;
- + Thực hiện tương tự duyệt cây tiền thứ tự với DFS, stack không chỉ lưu trữ các node, mà còn gán thêm vào mỗi node đó một trị số, trị số này chính là số lượng node đến thời điểm đang xét.

Độ phức tạp:

- **Cách 1:** O(N) với N là số node.

- **Cách 2:** O(N) với N là số node.
- **Cách 3:** O(N) với N là số node.

Code C++:

- **Cách 1:**

```
int count = -1;
int MAX = INT_MIN;

int maxDepth (node *root, int count)
{
    if (root == nullptr) {
        MAX = MAX > count ? MAX : count;
        if (MAX > 0) return MAX;
        return 0;
    }
    maxDepth(root -> left, count + 1);
    maxDepth(root -> right, count + 1);
}
```

- **Cách 2:**

```
int maxDepth(node* root) {
    if (root == nullptr) {
        return 0;
    }
    int left = maxDepth(root -> left);
    int right = maxDepth(root -> right);
    return max(left, right) + 1;
}
```

- **Cách 3:**

```
int maxDepth1(node *root)
{
    if (root == nullptr) {
        return 0;
    }
    stack<pair<node*, int>> st;
    st.push({root, 0});
    int res = 0;
    while(!st.empty())
    {
        pair<node*, int> top = st.top(); st.pop();
        res = max(res, top.second);
        if (top.first -> right != nullptr) {
            st.push({top.first -> right, top.second + 1});
        }
        if (top.first -> left != nullptr) {
            st.push({top.first -> left, top.second + 1});
        }
    }
}
```

```

    }
    return res;
}

```

Ví dụ 2:

Tồn tại hay không một đường đi từ gốc đến lá sao cho độ dài đường đi bằng số nguyên **targetSum** cho trước. Nếu tồn tại, trả về true, ngược lại trả về false.

Lời giải:

Ý tưởng:

- Duyệt DFS tương tự như duyệt cây bằng ngăn xếp và vòng lặp;
- Sau mỗi câu lệnh if (lệnh này để check xem node tiếp theo có phải nullptr hay không), nếu node tiếp theo không phải nullptr, thì bổ sung block else với mục đích kiểm tra điều kiện độ dài đường đi hiện tại có bằng targetSum hay không, nếu có trả về true, ngược lại tiếp tục chương trình.

Độ phức tạp: O(N) với N là số node.

Code C++:

```

bool ExistPath (node *root, const int targetSum)
{
    if (root == nullptr) {
        return false;
    }
    stack<pair<node*, int>> st;
    st.push({root, 0});
    while(!st.empty())
    {
        pair<node*, int> top = st.top(); st.pop();
        if (top.first -> right != nullptr) {
            st.push({top.first -> right, top.second + 1});
        }
        else {
            if (top.second == targetSum) {
                return true;
            }
        }
        if (top.first -> left != nullptr) {
            st.push({top.first -> left, top.second + 1});
        }
    }
}

```

```

        else {
            if (top.second == targetSum) {
                return true;
            }
        }
    }
    return false;
}

```

Ví dụ 3:

Tồn tại hay không một đường đi từ gốc đến lá sao cho tổng giá trị các node trên đường đi đó bằng số nguyên **targetSum** cho trước. Nếu tồn tại, trả về **true**, ngược lại trả về **false**.

Lời giải:

Ý tưởng:

- Sử dụng DFS tương tự như duyệt cây nhị phân bằng đệ quy;
- Bài toán được xây dựng hoàn toàn giống với ví dụ 1 (tìm độ sâu của cây nhị phân): mỗi node sẽ mang trong mình 1 giá trị (true hoặc false), nếu 2 cây con trái và phải của node đó đều nullptr, thì kiểm tra tổng giá trị đến thời điểm hiện tại, trả về true nếu tổng bằng **targetSum**, ngược lại trả về false. Nếu 1 trong 2 cây con vẫn chưa nullptr, ta tiến hành xét tiếp cây con chưa nullptr đó.
- Duyệt mọi trường hợp nhỏ trong từng cây con, chỉ cần có 1 cây con thỏa mãn điều kiện và trả về true, thì toàn bộ bài toán sẽ có giá trị true.

Độ phức tạp: O(N) với N là số node.

Code C++:

```

bool dfs(node *root, int targetSum, int count)
{
    if (root == nullptr) {
        return false;
    }

```

```

if (root -> left == nullptr && root -> right == nullptr) {
    return count + (root -> data) == targetSum;
}

bool left = dfs(root -> left, targetSum, count + root -> data);
bool right = dfs(root -> right, targetSum, count + root -> data);
return left || right;
}

bool existPath (node *root, int targetSum)
{
    return dfs(root, targetSum, 0);
}

```

Ví dụ 4:

Tìm tổng số node tốt trong cây nhị phân. Một node được gọi là tốt nếu như trên đường đi từ gốc đến node đó, không tồn tại node nào có giá trị lớn hơn node đó. Tích tập hợp các giá trị trên đường đi từ gốc đến node tốt là một dãy số không giảm.

Lời giải:

Ý tưởng:

- Sử dụng đệ quy DFS, duyệt qua tất cả các node trong cây, nếu gặp **node tốt** thì tăng giá biến đếm **ans** (biến đếm số lượng node tốt trong cây, là kết quả đề bài);
- Tuy nhiên, để xử lý biến **ans** từ trên xuống theo lời gọi đệ quy sẽ tốn nhiều công sức và chi phí, do đó ta tận dụng “sự hồi quy” sau mỗi lần gọi lệnh đệ quy để xử lý biến **ans** này, do đó biến **ans** sẽ được cập nhật từ dưới lên trên;
- Trong quá trình đệ quy DFS, ta duy trì biến **MAX** lưu trữ giá trị lớn nhất trên đường đi *cho đến thời điểm đang xét*. Duyệt đến cuối một nhánh nào đó, ta cũng tìm được giá trị lớn nhất trên đường đi từ gốc đến lá của nhánh đó;
- Lưu ý, mỗi lệnh gọi đệ quy DFS trả về 1 số nguyên, giá trị này chính là số lượng node tốt hiện tại; đó chính là giá trị của biến **ans**, được tính bằng tổng 2 số nguyên **left** và **right**;

- Ta kiểm tra tính tốt của node đó bằng cách so sánh giá trị của **MAX** (giá trị lớn nhất trên đường đi từ gốc đến node hiện tại) và **node -> data**, nếu **MAX** nhỏ hơn hoặc bằng thì tăng biến **ans** lên 1 đơn vị, đồng nghĩa với việc node đó là node tốt. Ngược lại, trả về giá trị **ans** chưa tăng giá trị cho lệnh gọi đệ quy hiện thời.

Độ phức tạp: O(N).

Code C++:

```
int goodNode (node *root, int MAX = INT_MIN)
{
    if (root == nullptr) {
        return 0;
    }
    int left = goodNode(root -> left, max(MAX, root -> data));
    int right = goodNode(root -> right, max(MAX, root -> data));
    int ans = left + right;
    if (root -> data >= MAX) {
        ans++;
    }
    return ans;
}
```

Ví dụ 5:

Tìm node tổ tiên chung gần nhất (*Lowest Common Ancestor*) của 2 node p, q cho trước trong một cây nhị phân.

Lời giải:

Ý tưởng:

- Khác với các bài ví dụ trước đó, bài toán này đê quy với 2 trường hợp cơ sở: *một* là trả về nullptr nếu toàn bộ cây con đang xét không chứa p, q; *hai* là trả về giá trị của root đó nếu root đang xét chính là p, q;
- Tại một node (giả sử là gốc) có 3 trường hợp xảy ra (lưu ý gốc ở đây là node có các nhánh con, không phải root của toàn bộ cây nhị phân):
 - + Nếu p hoặc q là gốc, thì kết quả không thể là các node bên dưới nút gốc;

- + Nếu p và q nằm ở 2 cây con khác nhau, thì kết quả chính là gốc;
 - + Nếu p và q nằm ở cùng 1 cây con, thì kết quả không phải là gốc.
- Để giải quyết bài toán, ta cần giải nhiều lần bài toán về 3 trường hợp trên, mỗi lần ứng với một node trong cây.
 - Để biết được node đang xét có chứa các node p, q hay không, ta cần duyệt cây bằng đệ quy DFS, nếu gặp p – q thì trả về p – q, nếu gặp nullptr thì trả về nullptr.
 - Như vậy nếu node đang xét có:
 - + **left** và **right** đều trả về giá trị khác null <TH2>, thì trả về node đó;
 - + **left != nullptr** thì trả về giá trị **left** <TH3>;
 - + **right != nullptr** thì trả về giá trị **right** <TH3>;
 - + <TH1> sẽ trở thành điều kiện dừng.

Độ phức tạp: O(N).

Code C++:

```
node* lowestCommonAncestor (node *root, node *p, node *q)
{
    if (root == nullptr) {
        return nullptr;
    }
    if (root == p || root == q) {
        return root;
    }

    node* left = lowestCommonAncestor(root -> left, p, q);
    node* right = lowestCommonAncestor(root -> right, p, q);

    if (left != nullptr && right != nullptr) {
        return root;
    }
    if (left != nullptr) {
        return left;
    }
    return right;
}
```

❖ **Ứng dụng BFS (breadth first search) vào cây nhị phân:**

Ví dụ 1:

Tìm tất cả các node tận cùng bên phải ở từng bậc (level) của cây nhị phân.

Lời giải:

Ý tưởng:

- Thực hiện tương tự BFS, sự khác biệt nằm ở các vòng lặp;
- Cách 1: <Phải xét điều kiện if>

Trong các lần lặp duyệt qua *từng node chung level* (*vòng lặp con phía trong*), nếu duyệt với vị trí cuối cùng, ta thực hiện việc in ra giá trị tại node đó, đó chính là node cần tìm;

- Cách 2: <Ngắn hơn và tối ưu hơn do bỏ xét điều kiện thừa>

Trong các lần lặp duyệt qua *từng level* (*vòng lặp cha phía ngoài*), trước khi bắt đầu vòng lặp duyệt qua từng node chung level (*vòng lặp con*), ta in ra giá trị ở cuối hàng đợi. Vì sau vòng lặp con trước đó, các node của level hiện tại được thêm vào theo thứ tự: node trái ở đầu hàng đợi, node phải ở cuối hàng đợi.

Độ phức tạp: $O(N.k)$ với N là số node và k là số node trong level đang xét.

Code C++:

Cách 1:

```
vector<node*> rightSideNode (node *root)
{
    vector<node*> res;
    if (root == nullptr) {
        return res;
    }
    queue<node*> qu;
    qu.push(root);
    while (!qu.empty()) {
        int N = qu.size();
        for (int i = 0; i < N; i++) {
            node *top = qu.front(); qu.pop();
            if (i == N - 1) {
                res.push_back(top);
            }
            if (top -> left != nullptr) {
                qu.push(top -> left);
            }
        }
    }
    return res;
}
```

```

        }
        if (top -> right != nullptr) {
            qu.push(top -> right);
        }
    }
    return res;
}

```

Cách 2:

```

vector<node*> rightSideNode (node *root)
{
    vector<node*> res;
    if (root == nullptr) {
        return res;
    }
    queue<node*> qu;
    qu.push(root);
    while (!qu.empty()) {
        int N = qu.size();
        res.push_back(qu.back());
        for (int i = 0; i < N; i++) {
            node *top = qu.front(); qu.pop();
            if (top -> left != nullptr) {
                qu.push(top -> left);
            }
            if (top -> right != nullptr) {
                qu.push(top -> right);
            }
        }
    }
    return res;
}

```

Ví dụ 2:

Tìm giá trị lớn nhất trên mỗi hàng cây. Quy định mỗi hàng cây là một tập hợp các node có cùng bậc (level).

Lời giải:

Ý tưởng:

- Cách 1:

- + Tương tự ví dụ 1 với các thao tác cơ bản của hàng đợi và vòng lặp. Ta cần lấy ra các node có giá trị lớn nhất ở mỗi hàng. Điều đó gợi ý ta cần sắp xếp các node ở cùng level theo một thứ tự nhất định và lấy ra giá trị lớn nhất đó. Với tần suất sắp xếp nhiều lần (ở mỗi hàng cây đều phải sắp xếp), cấu trúc dữ liệu hàng đợi ưu tiên là lựa chọn tối ưu nhất do độ phức tạp là $O(\log N)$ với thao tác Heapify (so với các thuật toán sắp xếp thông thường có độ phức tạp là $O(N \log N)$);
- + Tuy nhiên, ta không thay thế queue trong ví dụ 1 bằng priority_queue mà thêm một priority_queue vào thuật toán. Sau mỗi lần push một node vào queue, ta đồng thời cũng push node đó vào priority_queue. Kết quả là sau mỗi lần lặp lớn, ta vẫn đảm bảo thao tác BFS và cũng có cơ sở để truy xuất giá trị lớn nhất ở mỗi hàng;
- + Priority_queue được xây dựng bằng 2 cách:

- Sử dụng thông số bằng lamda function:

```
auto Compare = [] (node *a, node *b) -> bool
{
    return a->data < b->data;
};
```

- Sử dụng thông số bằng struct hoặc class (ở dạng public) có 1 hàm so sánh sử dụng nạp chồng toán tử gọi hàm `(_)`:

```
struct Compare1
{
    bool operator() (const node *a, const node *b)
    {
        return a->data < b->data;
    }
};
```

- Cách 2:

- + Mỗi vòng lặp con sẽ duyệt qua lần lượt tất cả các node trong cùng level, như vậy, ta cần duy trì một biến **MAX** lưu trữ giá trị của node lớn nhất, giá trị này được cập nhật trong mỗi lần vòng lặp con được thực hiện.
- + Sau khi vòng lặp for (con) thực hiện xong, biến **MAX** lúc đó sẽ giữ giá trị lớn nhất trong một hàng cây, push vào vector đầu ra, thực hiện cho đến khi vòng lặp while (cha) kết thúc.

Độ phức tạp:

- **Cách 1:** $O(N.(k+m))$ với N là số node trong cây, k là số node ở cùng level tại thời điểm đang xét, $m = k \rightarrow O(2.N.k)$;
- **Cách 2:** $O(N.k)$ với N là số node trong cây, k là số node ở cùng level tại thời điểm đang xét.

Code C++:

- **Cách 1:**

```
vector<node*> maxElement (node *root)
{
    vector<node*> res;
    queue<node*> qu;
    priority_queue<node*, vector<node*>, decltype(Compare)> pq(Compare);
    // hoặc: priority_queue<node*, vector<node*>, Compare1> pq;
    pq.push(root);
    qu.push(root);
    while (!qu.empty())
    {
        int N = qu.size();
        res.push_back(pq.top());
        while (!pq.empty()) {
            pq.pop();
        }
        for (int i = 0; i < N; i++) {
            node *top = qu.front(); qu.pop();
            if (top -> left) {
                qu.push(top -> left);
                pq.push(top -> left);
            }
            if (top -> right) {
                qu.push(top -> right);
                pq.push(top -> right);
            }
        }
    }
    return res;
}
```

- **Cách 2:**

```
vector<int> maxElement (node *root)
{
    vector<int> res;
    queue<node*> qu;
    qu.push(root);
```

```
while (!qu.empty())
{
    int N = qu.size();
    int MAX = INT_MIN;

    for (int i = 0; i < N; i++) {
        node *top = qu.front(); qu.pop();
        MAX = max(MAX, top -> data);
        if (top -> left) {
            qu.push(top -> left);
        }
        if (top -> right) {
            qu.push(top -> right);
        }
    }

    res.push_back(MAX);
}
return res;
}
```

D. CÁC BÀI TOÁN BỒ TRỢ:

1. Giải thuật quay lui (BackTracking):

https://youtu.be/efpaZznImN4?list=PLux-_phi0Rz0Hq9fDP4TlOulBl8APKp79

1. Định nghĩa:

- Thuật toán quay lui dùng để giải bài toán liệt kê các cấu hình, mỗi cấu hình được xây dựng bằng cách xây dựng từng phần tử, mỗi phần tử 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).

2. Cấu trúc chung cho bài toán quay lui:

```
BackTracking (int k) {  
    for ( i = <khả năng l> ; i <= <khả năng m> ; i++ )  
    {  
        if ( <chấp nhận khả năng i> ) {  
            X[k] = <khả năng i>;  
            if ( k == n )  
                <Thông báo cấu hình tìm được>;  
            else {
```

```

        BackTracking (k + 1);
    }
}

}

}

```

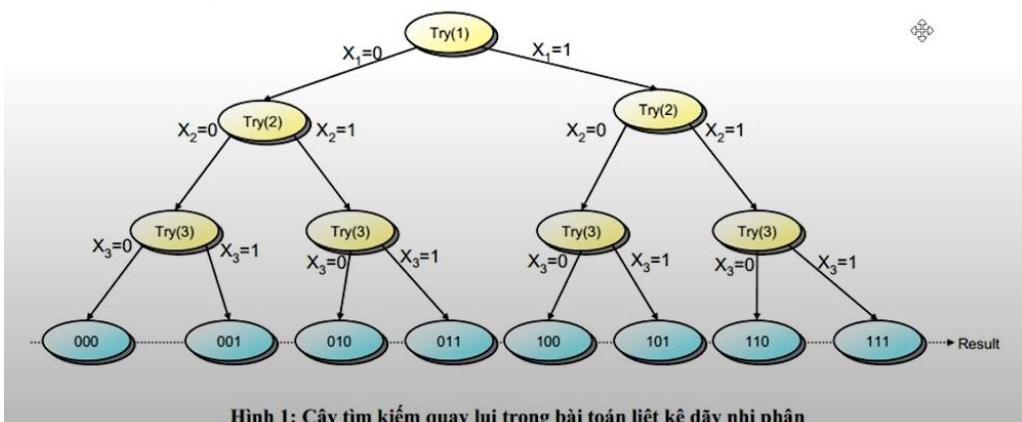
Nhân xét: BackTracking là một hàm đệ quy luôn thỏa mãn:

- Có thông số hàm là k (chỉ số của từng phần tử X);
- Luôn bắt đầu bằng vòng lặp for với i duyệt qua tất cả trường hợp có thể xảy ra của $X[k]$;
- Có phép gán $X[k] = i$ (cho phần tử $X[k]$ nhận tất cả các trường hợp khả dĩ i);
- Có câu lệnh rẽ nhánh if – else với:
 - + Nhánh if: điều kiện dừng (khi đã xét hết tất cả các khả năng) và in ra kết quả cấu hình;
 - + Nhánh else: đệ quy backTracking với $k + 1$;

3. Các bài toán kinh điển:

3.1 Quay lui tìm tất cả chuỗi bit nhị phân có độ dài N:

Ví dụ: Khi $n = 3$, cây tìm kiếm quay lui như sau:



```

#include <iostream>

using namespace std;

const int N = 3;
int X[3];

void print ()

```

```

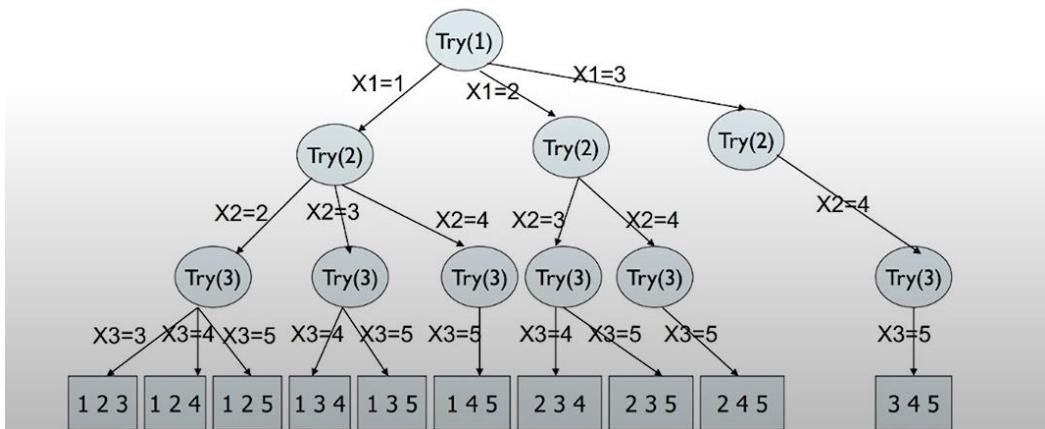
{
    for (int i = 1; i <= N; i++) {
        cout << X[i];
    }
    cout << endl;
}

void backTracking (int k)
{
    for (int i = 0; i <= 1; i++)
    {
        X[k] = i;
        if (k == N) {
            print();
        }
        else {
            backTracking (k + 1);
        }
    }
}

int main()
{
    backTracking (1);
    return 0;
}

```

3.2 Quay lui sinh tổ hợp chập K của N phần tử:



```

#include <iostream>

using namespace std;

const int N = 6, K = 4;
int X[K];

```

```

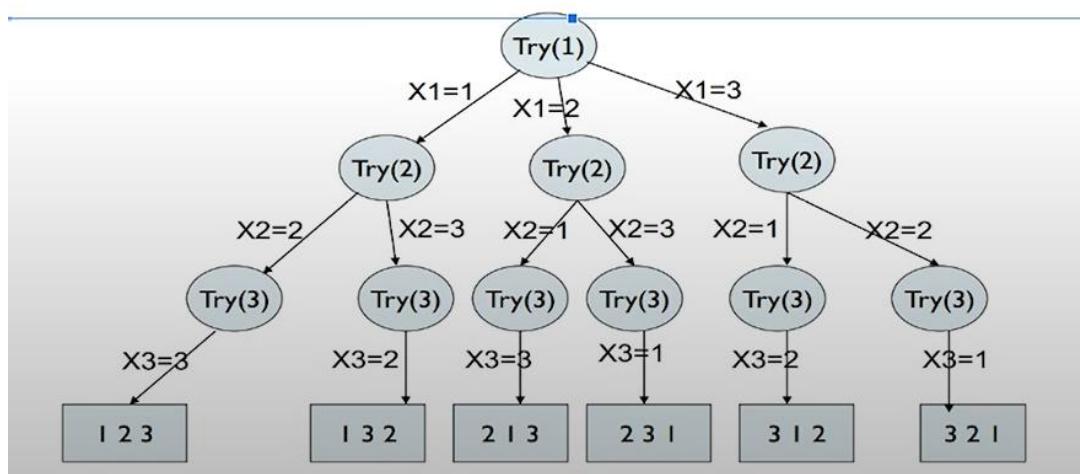
void print ()
{
    for (int i = 1; i <= K; i++) {
        cout << X[i];
    }
    cout << endl;
}

void backTracking (int k)
{
    for (int i = X[k - 1] + 1; i <= N - K + k; i++)
    {
        X[k] = i;
        if (k == K) {
            print();
        }
        else {
            backTracking (k + 1);
        }
    }
}

int main()
{
    backTracking (1);
    return 0;
}

```

3.3 Quay lui sinh hoán vị của N phần tử:



```

#include <iostream>

using namespace std;

const int N = 4;
bool isUsed[N] {false}; //Nếu khai báo dòng này sau int X[N] sẽ lỗi
int X[N];

void print ()
{
    for (int i = 1; i <= N; i++) {
        cout << X[i];
    }
    cout << endl;
}

void backTracking(int k)
{
    for (int i = 1; i <= N; i++)
    {
        if (isUsed[i - 1] == false)
        {
            isUsed[i - 1] = true;
            X[k] = i;
            if (k == N)
            {
                print();
            }
            else
            {
                backTracking(k + 1);
            }
            isUsed[i - 1] = false;
        }
    }
}

int main()
{
    backTracking (1);
    return 0;
}

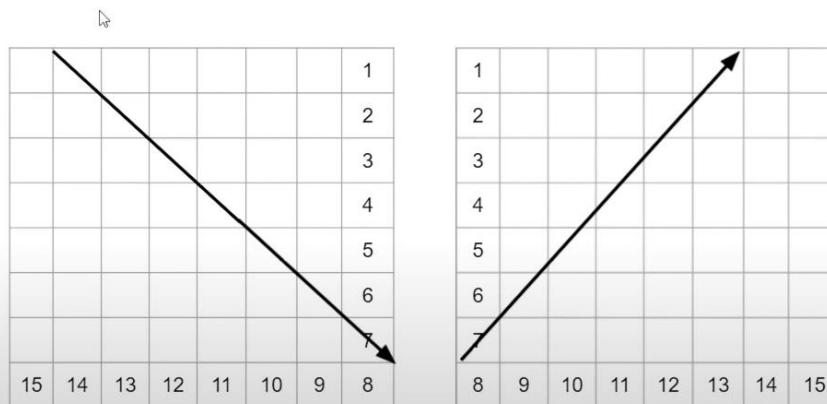
```

Lưu ý: Đây là bài toán sinh hoán vị của N phần tử liên tiếp bắt đầu từ 1 (1234, 1243,...) và để sinh hoán vị của N phần tử liên tiếp bắt đầu từ a = const, thì tại

vòng lặp for đầu tiên trong hàm BackTracking, ta cho i chạy từ a đến a + N – 1 ($i = a; i \leq a + N - 1$).

3.4 Quay lui bàn cờ vua:

Bài toán N queen: Tìm cách sắp xếp N quân hậu vào N hàng trên bàn cờ vua $N * N$ sao cho không có 2 quân hậu nào ánh nhau. Gọi $X = (X_1, X_2, \dots, X_N)$ là một nghiệm của bài toán, khi đó nếu $X_i = j$ thì có nghĩa ta xếp quân hậu hàng thứ i nằm ở cột j .



Các đường chéo xuôi được đánh chỉ số từ 1 tới $2n - 1$. Ô (i, j) bị đường chéo xuôi $i - j + n$ quản lý.

Các đường chéo ngược được đánh chỉ số từ 1 tới $2n - 1$. Ô (i, j) bị đường chéo ngược $i + j - 1$ quản lý.

```
#include <iostream>
#include <cstring>

using namespace std;

const int N = 8;
int X[100], col[100] {0}, d1[100] {0}, d2[100] {0};
int a[100][100];

void print ()
{
    memset (a, 0, sizeof(a));
    for (int i = 1; i <= N; i++) {
        a[i][X[i]] = 1;
    }

    for (int i = 1; i <= N; i++) {
        for (int j = 1; j <= N; j++) {
            cout << a[i][j];
        }
        cout << endl;
    }
}
```

```

        cout << endl;
    }

//Hang i cot j <=> Hang k cot j
void backTracking (int k)
{
    for (int j = 1; j <= N; j++)
    {
        if (col[j] == 0 && d1[k - j + N] == 0 && d2[k + j - 1] == 0) {
            X[k] = j;
            col[j] = 1;
            d1[k - j + N] = 1;
            d2[k + j - 1] = 1;
            if (k == N) {
                print();
            }
            else {
                backTracking (k + 1);
            }
        }
        //backTrack
        col[j] = 0;
        d1[k - j + N] = 0;
        d2[k + j - 1] = 0;
    }
}

int main()
{
    backTracking (1);
    return 0;
}

```

Độ phức tạp: <https://giaithuatlaptrinh.github.io/Quay-lui/>

2. Merge arrays – Trộn 2 mảng 1 chiều với các phần tử có giá trị không giảm:

Ý tưởng:

- Dùng 2 con trỏ a , b lần lượt quản lý 2 mảng đã cho, số lượng phần tử lần lượt là N , M (cũng gọi là kích thước dữ liệu đầu vào);
- Xét đồng thời các cặp phần tử (i, j) , nếu giá trị nào nhỏ hơn thì in ra giá trị đó;
 - + Nếu $a[i] \leq b[j]$ thì in ra $a[i]$, sau đó $++i$ để sang phần tử tiếp theo của mảng a ;
 - + Nếu $a[i] > b[j]$ thì in ra $b[j]$; sau đó $++j$ để sang phần tử tiếp theo của mảng b ;
 - + Tiếp tục cho đến khi 1 trong 2 (cả 2) không còn phần tử nào để xét, giả sử đến một lúc nào đó mảng a hết giá trị để so sánh thì ta chỉ cần in ra các giá trị còn lại của mảng b ;
 - + (Lưu ý: thay vì chỉ in ra các phần tử, ta có thể lưu các giá trị đã được sắp xếp vào mảng mới, giả sử mảng là tmpArr và chỉ số của mảng là count (giá trị ban đầu = 0)).

a	1	2	2	4					
b	3	5	6	7	9	10			
j									

$\text{tmpArr}[] = \{ \}$

a	1	2	2	4					
b	3	5	6	7	9	10			
j									

$\text{tmpArr}[] = \{ 1, 2 \}$

a	1	2	2	4					
b	3	5	6	7	9	10			
j									

$\text{tmpArr}[] = \{ 1, 2, 2 \}$

Code C++:

```
#include <iostream>
using namespace std;

int* mergeArr (int *a, int *b, int N, int M)
{
    int i = 0, j = 0;
    int count = 0;
    int *tmpArr = new int [N + M];
    while (i < N && j < M)
    {
        if (*(a + i) <= *(b + j)) {
            *(tmpArr + count++) = *(a + i++);
        }
        else {
            *(tmpArr + count++) = *(b + j++);
        }
    }
    while (i < N)
    {
        *(tmpArr + count++) = *(a + i++);
    }
    while (j < M)
    {
        *(tmpArr + count++) = *(b + j++);
    }
    return tmpArr;
}

void print (int *tmpArr, const int size) //size = N + M;
{
    for (int i = 0; i < size; i++) {
        cout << *(tmpArr + i) << " ";
    }
    delete[] tmpArr;
}

int main()
{
    const int N = 12;
    const int M = 15;
    int *a = new int [N] {1, 3, 5, 5, 6, 7, 8, 9, 12, 13, 14, 14};
    int *b = new int [M] {3, 4, 5, 6, 9, 10, 11, 23, 35, 46, 67, 79, 80, 81,
90};
    print(mergeArr(a, b, N, M), (N + M));
    delete[] a;
    delete[] b;
}
```

3. Thuật toán sinh tất cả các cấu hình (Generate All Configurations Algorithm):

Cấu trúc chung cho dạng toán:

```
bool allConfig = false;  
void print() {}  
void initialization() {}  
void generate() {}      //Sẽ làm cho allConfig = true nếu đã in ra tất cả cấu hình.  
  
int main () {  
    initialization();          //Khởi tạo cấu hình đầu tiên  
    while (!allConfig) {       //Nếu chưa in ra toàn bộ cấu hình  
        print();                //In ra cấu hình hiện tại  
        generate();              //Sinh cấu hình tiếp theo  
    }  
}
```

3.1 Bài toán sinh tất cả chuỗi bit có độ dài N:

Ưu điểm: Lợi thế của thuật toán sinh tất cả các cấu hình này có thể quyết định được đâu là cấu hình khởi đầu của chuỗi bit đó, khác với backTracking sinh ra tất cả chuỗi bit bắt đầu từ 00000...0000.

Ý tưởng:

- Khởi tạo: tạo ra cấu hình đầu tiên trong chuỗi;
- Sinh: sinh ra cấu hình tiếp theo;
- Cấu trúc chung:
 - *Khởi tạo cấu hình đầu tiên*
 - *While (khi chưa phải phần tử cuối cùng)*
 - *In ra cấu hình hiện tại;*
 - *Sinh cấu hình tiếp theo;*

Độ phức tạp: $O(N * 2^N)$ (in $O(N)$, sinh $O(N)$, while $O(2^N)$ do phải duyệt tối đa 2^N cấu hình)

Code C++:

```
#include <iostream>

using namespace std;

const int N = 4;
bool allConfig;
int X[N];

void print()
{
    for (int i = 1; i <= N; i++) {
        cout << X[i];
    }
    cout << endl;
}

void initialization()
{
    cout << "Nhap phan tu dau tien, gom " << N << " bit, moi bit phan biet
boi 1 khoang trang" << endl;
    for (int i = 1; i <= N; i++){
        cin >> X[i];
    }
}

void generate()
{
    int i = N;
    while (i >= 1 && X[i] == 1)
    {
        X[i] = 0;
        i--;
    }
    if (i == 0) {
        allConfig = true;
    }
    else {
        X[i] = 1;
    }
}

int main()
{
    initialization();
    allConfig = false;
    while (!allConfig) {
        if (true /*Them dieu kien neu can thiet*/) {
```

```

        print();
    }
    generate();
}
return 0;
}

```

3.2 Bài toán sinh tất cả tổ hợp chập K của N phần tử:

Ưu điểm: Lợi thế của thuật toán sinh tất cả các cấu hình này có thể quyết định được đâu là cấu hình khởi đầu của tổ hợp đó, khác với backTracking sinh ra tất cả tổ hợp bắt đầu từ 00000...0000.

Ý tưởng:

- Khác với backTracking, giải thuật sinh cấu hình tạo các cấu hình bắt đầu từ phần tử cuối cùng (backTracking xây dựng cấu hình từ phần tử đầu tiên);
- Chấp nhận rằng phần tử thứ i của một cấu hình nào đó chỉ có thể nhận giá trị tối đa là $N - K + i$, vì với mỗi tổ hợp ta chỉ in ra 1 thứ tự duy nhất (từ phần tử bé đến phần tử lớn) và bỏ qua các hoán vị còn lại của cấu hình đó;
- Cấu trúc chung:
 - Khởi tạo cấu hình đầu tiên;*
 - While (khi chưa là cấu hình cuối cùng)*
 - In ra cấu hình hiện tại;*
 - Sinh ra cấu hình tiếp theo;*

Độ phức tạp:

Code C++:

```

#include <iostream>
#include <ctime>

using namespace std;

const int N = 6, K = 4;
bool allConfig = false;
int X[K];

void print()

```

```

{
    for (int i = 1; i <= K; i++) {
        cout << X[i];
    }
    cout << endl;
}

void initialization()
{
    for (int i = 1; i <= K; i++) {
        X[i] = i;
    }
}

void generate ()
{
    int i = K;
    while (i >= 1 && X[i] == N - K + i) {
        i--;
    }
    if (i == 0) {
        allConfig = true;
    }
    else {
        X[i]++;
        for (int j = i + 1; j <= K; j++) {
            X[j] = X[j - 1] + 1;
        }
    }
}

void timeUsed (clock_t x, clock_t y)
{
    double time = (double) (y - x) / CLOCKS_PER_SEC;
    cout << "thoi gian: " << time << endl;
}

int main()
{
    clock_t start, end;
    start = clock();
    initialization();
    while(!allConfig) {
        print();
        generate();
    }
    end = clock();
    timeUsed(start, end);
}

```

```
    return 0;  
}
```

3.3 Bài toán sinh tất cả hoán vị của N phần tử:

Ý tưởng:

- Nhìn chung cách tiếp cận như backTracking, sự khác biệt nằm ở chỗ backTracking xây dựng từng phần tử trong một cấu hình từ đầu → cuối, trong khi generate all configurations xây dựng từ cuối → đầu.
- Khởi tạo cấu hình đầu tiên;
- Sinh cấu hình tiếp theo dựa trên cấu hình đầu tiên đó:
 - Đặc điểm cấu hình tiếp theo: cấu hình sau > cấu hình trước, nói đúng hơn là: cấu hình sau là số liền sau của cấu hình trước, không tồn tại một cấu hình nào vừa > cấu hình trước vừa < cấu hình sau. Ví dụ: 1243 → 1324 chứ không thể là: 1243 → 1342;
 - Cách tìm cấu hình tiếp theo đó:
 - Duyệt từ cuối → đầu (lưu ý thứ tự này là xuyên suốt);
 - Tìm cấu hình lớn hơn + tìm cấu hình nhỏ nhất (đã lớn hơn) = cấu hình lớn hơn nhỏ nhất (cấu hình tiếp theo);
 - Cấu hình lớn hơn: sẽ tồn tại nếu trong cấu hình hiện tại có một cặp phần tử ($X[i], X[j]$) ($i < j$ và $X[i] < X[j]$), ví dụ: 1243 sẽ có cấu hình lớn hơn là 1423, 1432, Tuy nhiên ta sẽ không hoán vị chúng với nhau, mà hoán vị ($X[i], X[k]$) với ($i < k$ và $X[i] < X[k]$ và $X[k]$ nhỏ nhất, tức không tồn tại $X[k']$ nào khác vừa > $X[i]$ vừa < $X[k]$;
 - Để làm được điều đó, ta cần nhận xét 1 điều rằng: sau khi đã tìm ra cặp phần tử ($X[i], X[j]$), thì từ từ vị trí $i + 1$ trở về sau sẽ là 1 dãy không tăng (giảm dần), như vậy để tìm $X[k]$ (số lớn hơn nhỏ nhất của $X[i]$) ta chỉ cần duyệt từ cuối → vị trí $i + 1$; tại vị trí đầu tiên nào có giá trị > $X[i]$ thì đó giá trị đó chính là $X[k]$, swap $X[i]$ với $X[k]$. Kết thúc bước này ta được cấu hình lớn hơn, tuy nhiên

đã lớn hơn chưa chắc đã nhỏ nhất, ta cần sang bước tiếp theo, nếu không có bước tiếp theo này, ta sẽ bị “miss” các cấu hình và sai kết quả;

- Cấu hình lớn hơn nhỏ nhất: để tìm cấu hình này, ta chỉ việc sắp xếp mảng từ $i + 1 \rightarrow N$ theo hướng tăng dần. Để làm điều này, có thể dùng kĩ thuật 2 con trỏ (left, right + while), hoặc hàm trong STL C++ (reverse);
- Cấu trúc chung:
 - *Khởi tạo cấu hình đầu tiên;*
 - *While (khi chưa là cấu hình cuối cùng)*
 - *In ra cấu hình hiện tại;*
 - *Sinh ra cấu hình tiếp theo;*

Độ phức tạp: $O(N!)$

Code C++:

```
#include <iostream>
#include <ctime>

using namespace std;

const int N = 3;
bool allConfig = false, isUsed[N] = {false};
int X[N];

void print()
{
    for (int i = 1; i <= N; i++) {
        cout << X[i];
    }
    cout << endl;
}

void initialization()
{
    for (int i = 1; i <= N; i++) {
        X[i] = i;
    }
}
```

```

void generate ()
{
    int i = N - 1;
    while (i >= 1 && X[i] > X[i + 1]) {
        i--;
    }
    if (i == 0) {
        allConfig = true;
    }
    else {
        int j = N;
        while (X[i] > X[j]) {
            j--;
        }
        swap(X[i], X[j]);

        int left = i + 1;
        int right = N;
        while (left <= right) {
            swap(X[left], X[right]);
            left++, right--;
        }
    }
}

void timeUsed (clock_t x, clock_t y)
{
    double time = (double) (y - x) / CLOCKS_PER_SEC;
    cout << "thoi gian: " << time << endl;
}

int main()
{
    clock_t start, end;
    start = clock();
    initialization();
    while(!allConfig) {
        print();
        generate();
    }
    end = clock();
    timeUsed(start, end);
    return 0;
}

```

3.4 Bài toán sinh phân hoạch của một số tự nhiên N (phân tích N thành tổng các số tự nhiên):

Ý tưởng:

- Mỗi tổ hợp các số hạng trong tổng của một phân hoạch là một cấu hình, mỗi cấu hình được lưu vào một mảng, cấu hình mới vẫn được lưu trong mảng đó bằng cách ghi đè giá trị lên cấu hình cũ (sau khi đã in cấu hình cũ ra, tương tự như các bài toán trước).
 - Ví dụ ($6 =$) $4 + 1 + 1$:
 - Đây là nội dung hàm generate()
 - Duyệt từ cuối → đầu mảng để tìm phần tử đầu tiên từ phải sang mà có giá trị $X[i]$ khác 1 (Các giá trị bằng 1 thì bỏ qua), giảm giá trị đó xuống 1 đơn vị, tức $X[i]--$; *Ở đây thì $X[1] = 4$ là giá trị cần tìm, sau đó giảm 1 đơn vị, tức $X[1] = 3$, lúc này mảng chỉ còn 1 phần tử là 3;*
 - Đếm số lượng chữ số 1 bị bỏ qua (tạm gọi M), bao gồm cả số 1 bị giảm đi vừa rồi, số lượng M này chính là phần cần bù thêm vào $X[i]$ tìm được ở trên để thu được N; *Ở đây thì có 2 số 1 trong mảng bị bỏ qua (mảng ban đầu 4-1-1), và thêm 1 chữ số 1 vừa bị giảm đi trong bước trên, nên số lượng chữ số 1 bị bỏ qua là 3 ($M = 3$);*
 - Chia lấy nguyên M cho $X[i]$ thu được kết quả Q; *Ở đây thì $Q = 3/3 = 1$;*
 - Chia lấy dư M cho $X[i]$ thu được kết quả R; *Ở đây thì $R = 3 \% 3 = 0$;*
 - Nếu tồn tại Q ($Q \neq 0$) thì đẩy Q lần giá trị $X[i]$ vào mảng; *Ở đây thì đẩy $Q = 1$ lần chữ số $X[i] = 3$ vào mảng, thu được mảng 3-3, lúc này mảng có 2 phần tử;*
 - Nếu tồn tại R ($R \neq 0$) thì đẩy 1 lần R vào mảng; *Ở đây thì $R = 0$ nên không đẩy bất kì phần tử nào vào mảng, lúc này mảng vẫn có 2 phần tử 3-3;*
 - Vòng lặp này thực hiện cho đến khi tất cả các phần tử trong mảng đều có giá trị 1, tức là toàn bộ phân hoạch (cấu hình) đều đã được duyệt, bài toán kết thúc; *Ở đây thì bài toán kết thúc khi mảng trở thành 1-1-1-1-1-1.*
 - Cấu trúc chung:
 - *Khởi tạo cấu hình đầu tiên;*
 - *While (khi chưa là cấu hình cuối cùng)*

- *In ra cấu hình hiện tại;*
- *Sinh ra cấu hình tiếp theo;*

Độ phức tạp:

Code C++:

Lưu ý: Nếu khai báo $X[N]$ sẽ in ra cấu hình cuối cùng không phải là 111...111 mà là 1, sở dĩ như vậy là vì chúng ta xét chỉ số mang từ 1 đến N , phần tử tại chỉ số 0 không xét đến, nên cần ít nhất $N + 1$ phần tử trong mảng, do đó phải khai báo $X[N + 1]$.

```
#include <iostream>

using namespace std;

const int N = 5;
int X[N + 1], cnt;
bool allConfig = false;

void print()
{
    for (int i = 1; i <= cnt; i++) {
        cout << X[i] << " ";
    }
    cout << endl;
}

void initialization()
{
    cnt = 1;
    X[cnt] = N;
}

void generate()
{
    int i = cnt;
    while (i >= 1 && X[i] == 1) {
        i--;
    }
    if (i == 0) {
        allConfig = true;
    }
    else {
        X[i]--;
        int number1s = cnt - i + 1;
        cnt = i;
        int q = number1s / X[i];
        int r = number1s % X[i];
    }
}
```

```
    if (q) {
        for (int j = 1; j <= q; j++) {
            cnt++;
            X[cnt] = X[i];
        }
    }
    if (r) {
        cnt++;
        X[cnt] = r;
    }
}

int main()
{
    initialization();
    while (!allConfig) {
        print();
        generate();
    }
}
```

Một số bài toán dùng giải thuật sinh cấu hình:

1. Sinh chuỗi bit tiếp theo của chuỗi bit đã cho: ví dụ 011 → 100.
2. Sinh c

E. KHÁI NIỆM KHÁC TRONG C++:

I. Template:

1. Tổng quan:

Template là một tính năng trong ngôn ngữ lập trình C++, được sử dụng để tạo ra các lớp, hàm hoặc kiểu dữ liệu tổng quát (generic) có thể hoạt động với nhiều kiểu dữ liệu khác nhau mà không cần viết lại mã.

Khi sử dụng template, bạn có thể tạo ra các lớp hoặc hàm mà không cần xác định trước kiểu dữ liệu cụ thể mà nó sẽ hoạt động. Thay vào đó, bạn chỉ cần chỉ định kiểu dữ liệu trong quá trình sử dụng template, và mã sẽ được biên dịch thành các phiên bản cụ thể tương ứng với kiểu dữ liệu bạn đã chỉ định.

Template giúp tăng tính linh hoạt và tái sử dụng mã trong C++, và nó là một tính năng quan trọng trong việc viết các chương trình tổng quát và hiệu quả.

2. Ví dụ:

Ví dụ 1: Lớp đa giác (Polygon).

Yêu cầu: Viết một lớp template để đại diện cho một đa giác. Lớp này sẽ có các thuộc tính và phương thức sau:

Thuộc tính:

- Mảng chứa các điểm trong đa giác.

Phương thức:

- *addPoint():* Thêm một điểm mới vào đa giác.
- *getPerimeter():* Tính và trả về chu vi của đa giác.
- *getArea():* Tính và trả về diện tích của đa giác.

Lời giải:

Code C++:

```
#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

template <typename C>
struct Point
{
    C x, y;
};

template <typename T, typename T1>
class Polygon
{
    vector<Point<T>> arr;
public:
    void print() const;
    void addPoint(void);
    T1 getPerimeter() const;
    T1 getArea() const;
};

template <typename T, typename T1>
void Polygon<T, T1>::print() const
{
    for (Point<T> i : arr)
    {
        cout << "(" << i.x << ", " << i.y << ")" << endl;
    }
    cout << endl;
}

template <typename T, typename T1>
void Polygon<T, T1>::addPoint()
{
    T a, b;
    cout << "Nhập: ";
    cin >> a >> b;
    arr.push_back({a, b});
}

template <typename T, typename T1>
T1 Polygon<T, T1>::getPerimeter() const
{
    T1 res = 0;
    vector<T1> tmp;
```

```

        for (size_t i = 0; i < arr.size(); i++)
    {
        if (i == arr.size() - 1) {
            T x1 = arr[i].x;
            T y1 = arr[i].y;
            T x2 = arr[0].x;
            T y2 = arr[0].y;
            tmp.push_back( sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2)) );
        }
        break;
    }
    T x1 = arr[i].x;
    T y1 = arr[i].y;
    T x2 = arr[i + 1].x;
    T y2 = arr[i + 1].y;
    tmp.push_back( sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2)) );
}
for (T1 i : tmp) {
    static int cnt = 1;
    cout << "Canh " << cnt << ":" << i << endl;
    cnt++;
    res += i;
}
cout << endl;
return res;
}

template <typename T, typename T1>
T1 Polygon<T, T1>::getArea() const
{
    cout << "Don't know how to calculate." << endl;
}

int main()
{
    Polygon<int, double> P;
    for (int i = 1; i <= 4; i++) {
        P.addPoint();
    }
    P.print();
    cout << "Chu vi: " << P.getPerimeter();
}

```

II. Nạp chồng toán tử (Operator Overloading):

1. Tổng quan:

- Nạp chồng toán tử là 1 trong nhiều tính năng cực kì phổ biến trong ngôn ngữ lập trình C++, cho phép người dùng tái định nghĩa (hoặc bổ sung định nghĩa) về cách sử dụng các toán tử có sẵn để thực hiện các chức năng khác.
- Ưu điểm:
 - + Trực quan hóa mã nguồn;
 - + Ngắn gọn và súc tích trong việc triển khai sử dụng;
 - + Hỗ trợ nhiều thuật toán;
 - + ...

2. 08 quy tắc nạp chồng toán tử: (2 buộc – 3 cấm)

- Chỉ có thể nạp chồng toán tử theo kiểu của nó:
 - + *Toán tử 1 ngôi không thể được nạp chồng theo kiểu của toán tử 2 ngôi;*
- Chỉ có thể nạp chồng toán tử trong các lớp (classes) mà người dùng tự định nghĩa:
 - + *Không thể nạp chồng toán tử cho các kiểu dữ liệu cơ bản có sẵn như int, float, double, char... hay string.h dưới dạng nạp chồng toàn cục;*
 - + *Các toán tử (+, -, *, /, ...) được tạo ra với chức năng cố định, không thể nạp chồng toàn cục để thay đổi chức năng đó, chỉ có thể nạp chồng chúng với các kiểu dữ liệu được tạo ra trong class mình tự định nghĩa;*
 - + *Có nghĩa rằng:*
 - *Int, float, double, ...: không thể nạp chồng toàn cục, nếu muốn nạp chồng toàn cục, thì cần phải có tham số (parameter) là đối tượng thuộc các lớp nhất định;*
 - *Int, float, double, ...: có thể nạp chồng khi chúng thuộc phạm vi một lớp tự định nghĩa.*

- Không thể dùng nạp chồng để thay thế chức năng vốn có của toán tử (hệ quả của ý số 3);
- Không thể nạp chồng cho các toán tử không thuộc ngôn ngữ C++;
- Không thể thay đổi tính kết hợp (associativity) và độ ưu tiên (precedence) của toán tử:
 - + *Tính kết hợp:* là thứ tự thực thi cần suy xét trong trường hợp các toán tử có cùng độ ưu tiên (same precedence). Ví dụ: $a + b + c$ thì được thực hiện từ trái sang phải;
 - + *Độ ưu tiên:* là thứ tự thực thi cần suy xét trong trường hợp các toán tử không cùng độ ưu tiên (different precedence). Ví dụ: $a + b * c$ thì thực hiện phép * trước, rồi mới đến phép +.
- Số lượng đối số cụ thể (explicit arguments) cần truyền vào hàm nạp chồng toán tử 01 ngôi:
 - + 0: nếu được nạp chồng bằng hàm thành viên (member function);
 - + 1: nếu được nạp chồng bằng hàm bạn (friend function);
- Số lượng đối số cụ thể (explicit arguments) cần truyền vào hàm nạp chồng toán tử 02 ngôi:
 - + 1: nếu được nạp chồng bằng hàm thành viên (member function);
 - + 2: nếu được nạp chồng bằng hàm bạn (friend function);
- Trong trường hợp không thể nạp chồng toán tử bằng hàm bạn, hãy chuyển sang nạp chồng bằng hàm thành viên.

3. Các loại nạp chồng toán tử phổ biến:

Theo nội dung phần số 2 (08 quy tắc nạp chồng toán tử), ta có 02 phương pháp triển khai hàm nạp chồng:

- Nạp chồng thông qua hàm thành viên;
- Nạp chồng thông qua hàm bạn.

2.1 Nạp chồng toán tử một ngôi (Unary Operator Overloading):

2.2 Nạp chồng toán tử hai ngôi (toán tử nhị phân) (Binary Operator Overloading):

- Thông qua hàm thành viên:

```
#include <iostream>

using namespace std;

class Try
{
    int data;
public:
    Try(int a) : data(a) {}
    int operator+ (const int &);
    int operator+ (const Try &);
};

int Try::operator+ (const int &arg2)
{
    return data + arg2;
}

int Try::operator+ (const Try &agr2)
{
    return data + agr2.data;
}

int main()
{
    Try arg(4);
    int sum1 = arg + 5;
    cout << "Sum1: " << sum1 << endl;

    Try arg1(5), arg2(7);
    int sum2 = arg1 + arg2;
    cout << "Sum2: " << sum2 << endl;
}
```

```
#include <iostream>

using namespace std;

class Try
{
    int data;
public:
    Try(const int a) : data(a) {}
    Try operator+ (const int &);
    Try operator+ (const Try &);
```

```

void print() const{
    cout << data << endl;
}
};

Try Try::operator+ (const int &arg2)
{
    return Try(data + arg2);
}

Try Try::operator+ (const Try &agr2)
{
    return Try(data + agr2.data);
}

int main()
{
    Try arg(4);
    Try sum1 = arg + 5;
    cout << "Sum1: "; sum1.print();

    Try arg1(5), arg2(7);
    Try sum2 = arg1 + arg2;
    cout << "Sum2: "; sum2.print();
}

```

Theo quy tắc số 07, hàm nạp chồng toán tử 2 ngôi có 1 đối số.

- **Thông qua hàm bạn:**

2.3 Nạp chồng toán tử gán (Assignment Operator Overloading):

2.4 Nạp chồng toán tử gọi hàm chuyển đổi kiểu (Type Conversion Operator Overloading):

2.5 Nạp chồng toán tử gọi hàm chức năng (Function Call Operator Overloading):

III. Ivalue và Rvalue – Ivalue Reference và Rvalue Reference:

Ivalue (left value, locator value):

- là *thực thể* (*biến, hàm, biểu thức toán học...*) có giá trị cụ thể (hoặc không)
- có vùng nhớ xác định
- có thể gán giá trị cho nó
- lưu trữ giá trị đó trong khoảng thời gian dài (sẽ biến mất khi tầm vực của nó kết thúc)
- bền lâu

Rvalue (right value):

- là *thực thể* (*hàm, biểu thức toán học...*) buộc có giá trị cụ thể
- có vùng nhớ xác định
- không thể gán trị cho nó (tức chỉ có 1 giá trị duy nhất, xác định)
- giá trị này chỉ có thể tồn tại khoảng thời gian ngắn (thường sẽ biến mất ngay sau khi câu lệnh chứa nó thực hiện xong)
- tạm thời

Lvalue reference (phép tham chiếu đến lvalue):

- là một dạng phép gán
- là một phép tham chiếu (tham khảo) đến một lvalue
- lvalue reference dùng để tham chiếu đến một lvalue, nhưng không thể dùng để tham chiếu đến một rvalue.
- Nếu muốn tham chiếu đến một rvalue từ một lvalue, thì buộc dùng const lvalue reference.

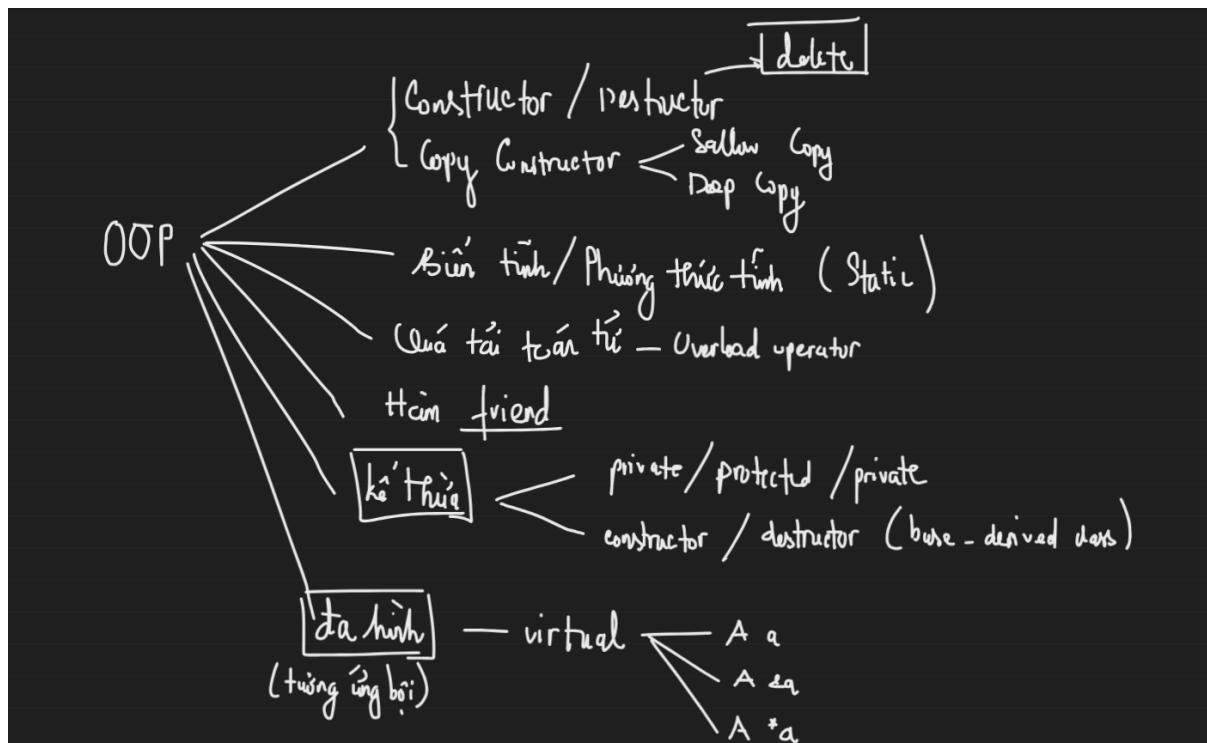
```
// Đúng: phép gán thông thường, trong đó (a: lvalue) và (2: rvalue)
int a = 2;
// Đúng: lvalue reference thực hiện tham chiếu từ lvalue b đến lvalue a
int &b = a;
// Sai: lvalue reference thực hiện tham chiếu từ lvalue c đến rvalue 2
int &c = 2;
const int &d = 2; // Đúng: const lvalue reference thực hiện tham chiếu từ
const lvalue đến rvalue 2
```

Rvalue reference (phép tham chiếu đến rvalue):

- là một dạng phép gán
- là một phép tham chiếu (tham khảo) đến một rvalue
- chỉ có rvalue reference mới có thể tham chiếu đến rvalue

F. LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG (OOP):

Trong phần này chỉ đề cập đến những lưu ý cần nhớ, những điều quan trọng mà trong lập trình hướng đối tượng cần sử dụng đến. Tuy cơ bản nhưng cấp thiết.



1. Sơ khai:

- OOP là một mô hình lập trình được phát triển trong C++, không có trong C;
- OOP được xây dựng bằng cấu trúc dữ liệu lớp **class**;
- Về class:
 - + Gồm 2 phần: thuộc tính (attribute) và phương thức (method) hay còn gọi là biến thành viên và hàm thành viên;
 - + Biến mô tả đặc tính của đối tượng, hàm mô tả hành vi của đối tượng. Biến thường được gắn thẻ phạm vi truy cập private, hàm thường được gắn thẻ phạm vi truy cập public;
 - + Hàm được xem là trung gian thực hiện công tác giao tiếp giữa giao diện người dùng và các đặc điểm riêng tư của chương trình;
 - + Một class hoàn chỉnh là đơn vị cấu thành một đối tượng trong tự nhiên;

2. Hàm khởi tạo và Hàm hủy (Constructor và Destructor):

a) Hàm khởi tạo – Constructor:

b) Hàm hủy – Destructor:

- Đối tượng sẽ được hủy khi thỏa 1 trong 2 điều kiện sau:
 - + Đối tượng nằm ngoài phạm vi đoạn code mà chương trình đang chạy trên đó (ví dụ chạy hết hàm main(), ...) <Xảy ra tự nhiên khi đối tượng được cấp phát tĩnh>;
 - + Người lập trình chủ động xóa đối tượng bằng từ khóa **delete**. <Xảy ra khi có sự tác động từ người lập trình, lúc này hàm hủy của class được gọi trước khi giải phóng vùng nhớ do đối tượng quản lý>.
- Trong OOP, hàm hủy và toán tử delete có mối liên hệ khá mật thiết với nhau khi ta khai báo đối tượng dưới dạng con trỏ và quản lý trong bộ nhớ HEAP. Cụ thể, khi ta cấp phát động đối tượng bằng toán tử new như ví dụ về đối tượng a như sau:

```
class A
{
    int a;
public:
    A() : a(0) {}
    ~A() {} // Hàm hủy của lớp A
};

int main()
{
    A *a = new A(); // Khởi tạo đối tượng A trên heap
    int *b = new int; // Khởi tạo một con trỏ int trên heap

    delete a; // Giải phóng đối tượng A và gọi hàm hủy ~A()
    delete b; // Giải phóng vùng nhớ của con trỏ int

    return 0;
}
```

- Cơ chế hoạt động của 2 lệnh delete trên là khác nhau:
 - + Delete a: gọi hàm hủy của lớp A trước, sau đó mới giải phóng vùng nhớ do a quản lý;
 - + Delete b: đơn thuần giải phóng vùng nhớ do b quản lý.

- Điều đó có nghĩa: khi khai báo đối tượng bằng cấp phát động, ta hoàn toàn chịu trách nhiệm quản lý (khai báo và hủy) vùng nhớ trong bộ nhớ HEAP, đối tượng và vùng nhớ do nó quản lý chỉ được giải phóng khi người dùng chủ động sử dụng **delete** phù hợp (chương trình tiến hành gọi hàm hủy của lớp mà đối tượng thuộc về, sau đó giải phóng vùng nhớ mà đối tượng đang quản lý). Nếu không **delete**, khả năng rò rỉ bộ nhớ (memory leak) sẽ rất cao khiến chương trình không còn đủ bộ nhớ cho các lần thực thi tiếp theo.
- Để khắc phục điều này, hãy xem mô hình RAII tại đây: <https://s.net.vn/5QBW>
- *Quy tắc: Nếu class của bạn cần cấp phát bộ nhớ động, hãy sử dụng mô hình RAII, chử đừng nên cấp phát động trực tiếp cho những đối tượng của class của bạn.*

3. Hàm bạn và Lớp bạn (Friend function và Class function):

1. Hàm bạn – friend function:

- Hàm bạn cần chứa 1 thông số là đối tượng của lớp cần làm bạn. Vì hàm bạn về bản chất không có sự khác biệt với hàm thông thường, ngoại trừ việc có thể truy cập vào các biến và hàm thành viên thuộc phạm vi truy cập private của lớp đó. Nếu không chứa thông số đó, ta sẽ không có cơ sở để truy cập vào các biến thành viên private của lớp đang xét.
- Cần lưu ý rằng nên truyền tham chiếu cho đối tượng để có thể thay đổi trạng thái hiện tại của đối tượng theo mong muốn, nếu không sẽ không có sự thay đổi nào diễn ra.
- Việc truyền tham chiếu cho đối số của hàm bạn cũng là điều bắt buộc, vì khi đó ta đang thao tác với chính các thuộc tính (attributes) của lớp, thay vì tạo một bản sao của thuộc tính (qua việc tạo bản sao của đối tượng). Nếu không truyền tham chiếu, một bản sao sẽ được tạo ra, một đối tượng mới cũng theo đó được tạo. Đối tượng này mang đầy đủ chức năng của đối tượng đang tham trị (đối tượng chính) đến, tức cũng có hàm hủy, và do đó hàm hủy được thực thi thêm một lần nữa. Điều này dẫn đến có thể làm mất các thuộc tính của lớp sau khi hàm hủy được gọi (lúc hàm bạn đã thực thi xong). Các thao tác sau với thuộc tính này sau khi hàm bạn được thực thi cũng sẽ bị ảnh hưởng, có thể dẫn đến sai kết quả. (Ví dụ: thuộc tính được cấp phát động, trong hàm hủy giải phóng đi vùng nhớ của thuộc tính đó, do đó nếu không truyền tham chiếu, hàm hủy sẽ giải phóng đi vùng nhớ của thuộc tính nói trên, làm thay đổi nội dung của nó, khiến cho các thao tác phía sau bị ảnh hưởng).

```
class Accumulator
{
private:
    int m_value;
public:
    Accumulator() { m_value = 0; }
    void add(int value) { m_value += value; }

    // Make the reset() function a friend of this class
    friend void reset(Accumulator &accumulator);
};
```

```

// reset() is now a friend of the Accumulator class
void reset(Accumulator &accumulator)
{
    // And can access the private data of Accumulator objects
    accumulator.m_value = 0;
}

int main()
{
    Accumulator acc;
    acc.add(5); // add 5 to the accumulator
    reset(acc); // reset the accumulator to 0

    return 0;
}

```

4. Khai báo biến thành viên (declaration) – khởi tạo giá trị biến thành viên (initialization) trong class:

a) Cạm bẫy (pitfall):

- Trong class A, không thể khai báo biến thành viên a trong *cùng lúc* với việc khởi tạo giá trị cho biến thành viên a đó. Ví dụ: ta không thể làm như sau:

```

class A
{
    int a = 4;
};

```

Điều đó đúng cho mọi kiểu dữ liệu của biến thành viên a, dù a có là một kiểu dữ liệu thông thường (int, float, char, ...) hay kiểu dữ liệu struct, class.

- Tuy nhiên, từ phiên bản C++11 trở đi, ta có thể thực hiện công việc khởi tạo giá trị cho biến thành viên cùng lúc với thời điểm nó được khai báo trong class *khi biến thành viên có kiểu dữ liệu cơ bản (int, float, char, ...)*; Với trường hợp có kiểu struct hoặc class, ta vẫn không thể thực hiện việc khai báo và khởi tạo cùng lúc.
- Điều đó có nghĩa:
 - + Đây là cách làm sai (tương tự cho trường hợp B là struct):

```

class B

```

```
{  
    int b;  
public:  
    B(int data) : b(data) {}  
};  
  
class A  
{  
    B a(5);  
};
```

- + Đây là cách làm đúng (tương tự cho trường hợp B là struct):

```
class B  
{  
    int b;  
public:  
    B(int data) : b(data) {}  
};  
  
class A  
{  
    B a;  
public:  
    A(int data) : a(data) {}  
};
```

- Nó tương tự như việc ta phải khai báo a có tham số khi class B không có default constructor (hàm khởi tạo không có tham số), và việc khai báo a có tham số sẽ phải được thực hiện trong constructor của class A, thay vì làm trực tiếp trong công tác khai báo.

các dạng bài tập

1. Tìm upper bound của một hàm $f(n)$ đã cho:

- $O(g(n)) = \{f(n) : \text{tồn tại các hằng số dương } c \text{ và } n_0 \text{ sao cho } 0 \leq f(n) \leq c * g(n) \text{ với mọi } n \geq n_0\}$.
- Khi đó $g(n)$ được gọi là tiệm cận trên của $f(n)$. $O(g(n))$ được gọi là upper bound của $f(n)$
- Tìm upper bound \Leftrightarrow tìm $O(g(n)) \Leftrightarrow$ chỉ ra c và n_0 thỏa $0 \leq f(n) \leq c * g(n)$ ($n \geq n_0$).
- Bài toán đơn giản là biến đổi các bất đẳng thức để tìm c và n_0 .

Bước 1: Xét hàm $f(n)$:

- Nếu là hàm bậc 1 (tuyến tính) dạng: $f(n) = an + b$ ($b > 0$), thì ta luôn có:

$$f(n) \leq (a+1)n \Leftrightarrow an + b \leq (a+1)n \Leftrightarrow an \geq b \Leftrightarrow n \geq \left\lceil \frac{b}{a} \right\rceil, \quad \text{tức là:}$$

$$\begin{cases} c = a + 1 \\ n_0 = \left\lceil \frac{b}{a} \right\rceil \end{cases}. \text{ Với } b < 0 \text{ thì } f(n) = an + b \leq an, \text{ khi đó } c = a \text{ và } n_0 = C \text{ bất kì.}$$

- Nếu là hàm bậc 2 dạng: $f(n) = an^2 + bn + c$, thì ta luôn có $f(n) \leq (a+1)n^2 + bn + c$, biến đổi tương tự như trên để tìm n_0 .
- Các dạng còn lại thực hiện tương tự.

Bước 2: Kết luận

- Chỉ ra $O(g(n))$
- Chỉ ra c và n_0

2. Tìm độ phức tạp (thời gian) của một thuật toán chứa vòng lặp:

Bước 1: Xét các câu lệnh đơn:

- Tất cả các câu lệnh gán, lựa chọn với các biểu thức, phép toán số học, luận lí cơ bản đều có **độ phức tạp O(1)**;
- Tất cả các câu lệnh xét điều kiện với các biểu thức, phép toán số học, luận lí cơ bản đều có **độ phức tạp O(1)**;

Bước 2: Xét các câu lệnh khởi tạo vòng lặp:

- Với vòng lặp đơn (chỉ gồm 1 vòng lặp):
 - + Nếu số lần lặp bằng n (lặp với số lần lặp độ lớn dữ liệu đầu vào), thì **độ phức tạp là O(n)**;
 - + Nếu số lần lặp bằng k << n hay k = const, thì **độ phức tạp quy về O(1)**.
- Với vòng lặp không đơn (gồm 2 vòng lặp chồng nhau trở lên)
 - + Nếu vòng lặp con (phía trong) có số lần lặp **độc lập** với vòng lặp cha (bên ngoài), thì xem từng vòng lặp là vòng lặp đơn rồi tính O. Khi đó, độ phức tạp tổng bằng tích các độ phức tạp từ các vòng lặp đơn.
 - + Nếu vòng lặp con (phía trong) có số lần lặp **phụ thuộc** vào vòng lặp cha (bên ngoài), thì vẽ bảng thể hiện số lần thực hiện vòng lặp cụ thể. Tùy thuộc vào bài toán rồi tính độ phức tạp tổng.

```
1 for (i = 0; i < n; i++)
2   for (j = 0; j < n; j++)
3     a[i][j] = 0;
```

Số lần lặp độc lập
(j không phụ thuộc i)

```
1 sum = 0
2 for (i = 0; i < n; i++)
3   for (j = i + 1; j <= n; j++)
```

Số lần lặp phụ thuộc
(j phụ thuộc i)


```

#include <iostream>

using namespace std;

void sortArr (int *ptr, const int N)
{
    for (int i = 0; i < N; i++)
    {
        for (int j = i + 1; j < N; j++)
        {
            if (*(ptr + i) > *(ptr + j))
            {
                swap (*(ptr + i), *(ptr + j));
            }
            else continue;
        }
    }
    return;
}

int findValue (int *ptr, const int N, const int value)
{
    int tmp = N/2 - 1;
    if (value > *(ptr + tmp))
    {
        return findValue ((ptr + tmp + 1), N - tmp, value);
    }
    else if (value < *(ptr + tmp))
    {
        return findValue (ptr, N/2, value);
    }
    else return *(ptr + tmp);
}

void printArr (int *ptr, int N)
{
    for (int i = 0; i < N; i++)
    {
        cout << ptr[i] << " ";
    }
    return;
}

int main()
{
    const int N = 9;
    int arr[N] = {1, 32, 65, 24, 19, 99, 43, 100, 34};
    int *ptr = arr;
}

```

```
sortArr (ptr, N);
printArr (ptr, N);
cout << endl;
cout << findValue (ptr, N, 19);
}
```

```

void reverseList (node *&head)
{
    if (head == nullptr || head -> next == nullptr) return;
    node *preNode = nullptr;
    node *curNode = head;
    node *nextNode;
    while (curNode != nullptr) {
        nextNode = curNode -> next;
        curNode -> next = preNode;
        preNode = curNode;
        curNode = nextNode;
    }
    head = preNode;
}

int main()
{
    node *head = nullptr;
    for (int i = 1; i <= 5; i++) {
        pushFront (head, i);
    }
    duyet(head);
    cout << endl << endl;

    pushIn(head, 432, 4);
    duyet(head);
    cout << endl << endl;

    for (int i = 100; i <= 105; i++) {
        pushBack (head, i);
    }
    duyet(head);
    cout << endl << endl;

    popFront (head);
    duyet(head);
    cout << endl;
    cout << "head: " << head -> data << endl << endl;

    popBack(head);
    duyet(head);
    cout << endl;
    cout << "head: " << head -> data << endl << endl;

    popIn(head, 3);
    duyet(head);
    cout << endl;
}

```

```
    cout << "head: " << head -> data << endl << endl;  
}
```



```
int count (node *head)
{
    int count = 0;
    node *tmpNode = head;
    while (tmpNode != nullptr) {
        count++;
        tmpNode = tmpNode -> next;
    }
    return count;
}

void pushFront (node *&head)
{
    node *tmpNode = head;
    node *newNode = createNode();
    if (head == nullptr) {
        head = newNode;
    }
    else {
        newNode -> next = tmpNode;
        head = newNode;
    }
}

void pushIn (node *&head, int index)
{
    node *tmpNode = head;
    node *newNode = createNode();
    if (head == nullptr) {
        head = newNode;
    }
    else if (index <= 0 || index > count(head)) {
        cout << "Invalid index" << endl;
        return;
    }
    else if (index == 1) {
        pushFront(head);
    }
    else {
        for (int i = 1; i <= index - 2; i++) {
            tmpNode = tmpNode -> next;
        }
        newNode -> next = tmpNode -> next;
        tmpNode -> next = newNode;
    }
}
```

```

void pushBack (node *&head)
{
    node *tmpNode = head;
    node *newNode = createNode();
    if (head == nullptr) {
        head = newNode;
    }
    else {
        while (tmpNode -> next != nullptr) {
            tmpNode = tmpNode -> next;
        }
        tmpNode -> next = newNode;
    }
}

void popFront (node *&head)
{
    node *tmpNode = head;
    if (head == nullptr) {
        cout << "Invalid request" << endl;
        return;
    }
    else {
        head = head -> next;
        delete tmpNode;
    }
}

void popIn (node *&head, int index)
{
    node *tmpNode = head;
    if (head == nullptr) {
        cout << "Invalid request" << endl;
    }
    else if (index <= 0 || index > count(head)) {
        cout << "Invalid index" << endl;
        return;
    }
    else if (index == 1) {
        popFront(head);
    }
    else {
        for (int i = 1; i <= index - 2; i++) {
            tmpNode = tmpNode -> next;
        }
        node *removedNode = tmpNode -> next;
        tmpNode -> next = tmpNode -> next -> next;
        delete removedNode;
    }
}

```

```
    }

}

void popBack (node *&head)
{
    node *tmpNode = head;
    if (head == nullptr) {
        cout << "Invalid request" << endl;
    }
    else if (tmpNode -> next == nullptr) {
        head = nullptr;
        delete tmpNode;
    }
    else {
        while (tmpNode -> next -> next != nullptr) {
            tmpNode = tmpNode -> next;
        }
        node *lastNode = tmpNode -> next;
        tmpNode -> next = nullptr;
        delete lastNode;
    }
}

int main()
{
    node *head = nullptr;
    pushFront(head);
    print(head);

    pushBack(head);
    print(head);

    pushBack(head);
    print(head);

    pushIn(head, 2);
    print(head);

    cout << "popFront" << endl;
    popFront(head);
    print(head);
    cout << "popIn" << endl;
    popIn(head, 1);
    print(head);
    cout << "popBack" << endl;
    popBack(head);
    print(head);
    cout << "Reverse" << endl;
```

```
reverseList(head);
print(head);

    delete head;
}
```

```
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>
#include <functional>

using namespace std;

void print (vector<int> arr)
{
    for (int i : arr) {
        cout << i << " ";
    }
    cout << endl;
}

int binarySearch (vector <int> arr, int left, int right, int value)
{
    if (left >= right) return -1;
    int mid = (left + right)/2;
    if (arr[mid] == value) {
        return mid;
    }
    else if (arr[mid] > value) {
        return binarySearch(arr, left, mid - 1, value);
    }
    else if (arr[mid] < value) {
        return binarySearch(arr, mid + 1, right, value);
    }
}

void top5 (vector <int> arr)
{
    if (arr.size() < 5) {
        cout << "There is not enough elements to print!" << endl;
        return;
    }
    for (vector<int>::reverse_iterator i = arr.rbegin(); i <= arr.rbegin() +
4; i++) {
```

```
        cout << *i << " ";
    }
    cout << endl;
}

void nhap (vector<int> &arr)
{
    int num;
    cout << "Nhập mang: " << endl;
    while (true) {
        cin >> num;
        if (num >= 0) {
            arr.push_back(num);
        }
        else break;
    }
}

int main()
{
    vector<int> arr;
    nhap(arr);
    int k;
    cout << "Nhập k: ";
    cin >> k;

    sort(arr.begin(), arr.end(), [] (const int &a, const int &b) -> bool
{return a < b;});
    print(arr);

    cout << "Địa chỉ tại k: " << binarySearch(arr, 0, arr.size() - 1, k) <<
endl;

    top5(arr);
}
```