EXTERN C++

PROGRAMMING LANGUAGE

- Programming Language: là ngôn ngữ dùng để diễn tả thuật toán sao cho máy tính hiểu và thực hiện được.
- 2 loại:
  - Ngôn ngữ lập trình bậc thấp (low-level programming language):
    - Mã máy;
    - Hợp ngữ (Assembly).
  - Ngôn ngữ bậc cao (high-level programming language):
    - FORTRAN (1956);
    - PASCAL (1970), C (1972), C++ (1980), Java (1995), C# (2001), Python (1990), Swift (2014), Kotlin (2017), …
  - **FRAMEWORK & SOFTWARE LIBRARIES**

NON-STRUCTURED PROGRAMMING

- Là phương pháp xuất hiện đầu tiên.
- Giải quyết các bài toán nhỏ, tương đối đơn giản (lĩnh vực tính toán).

```
10      k=1
20      gosub 100
30      if y > 120 goto 60
40      k = k + 1
50      goto 20
60      print k, y
70      stop
100     y = 3*k*k + 7*k - 3
110     return
```

Lệnh nhảy đến vị trí bất kỳ trong chương trình

- Đặc trưng:
  - Đơn giản;
  - Đơn luồng;
  - Sử dụng biến toàn cục, lạm dụng lệnh GOTO;
  - Chỉ gồm một chương trình chính.

PROCEDURE-ORIENTED PROGRAMMING (POP)

- Tổ chức chương trình thành các **chương trình con**
  - PASCAL: thủ tục & hàm, C: hàm.
- Chương trình hướng cấu trúc = cấu trúc dữ liệu + tập hợp hàm.
- Trừu tượng hóa chức năng (Functional Astraction):
  - Không quan tâm đến cấu trúc hàm;
  - Chỉ cần biết kết quả thực hiện của hàm.

→ Nền tảng của lập trình hướng cấu trúc.

PROCEDURE-ORIENTED PROGRAMMING (POP)

- Ví dụ:

```
int func(int j)
{
return (3*j*j + 7*j-3);
}
int main()
{
int k = 1
while (func(k) < 120)
k++;
printf("%d\t%d\n", k, func(k));
return(0);
}
```

NGÔN NGỮ C VÀ C++

- Ngôn ngữ C ra đời năm 1972.
- Phát triển thành C++ vào năm 1983.

KEYWORDS

- Một số từ khóa (keyword) mới đã được đưa vào C++ ngoài các từ khóa có trong C.
- Các chương trình sử dụng các tên trùng với các từ khóa cần phải thay đổi trước khi chương trình được dịch lại bằng C++.

asm	catch	class	delete	friend	inline
new	operator	private	protected	public	
template	this	throw	try	virtual	

COMMENT

- C:
  - Chú thích bằng /* ... */ → dùng cho khối chú thích lớn gồm nhiều dòng.
- C++: giống như C
  - Đưa thêm chú thích bắt đầu bằng // → dùng cho các chú thích trên một dòng.
- Ví dụ:

```
/*      Đây là
        chú thích trong C */
// Đây là chú thích trong C++
```

QUY TẮC ĐẶT TÊN

- Quy tắc Pascal: viết hoa chữ cái đầu tiên của mỗi từ
  - Method, Interface, Enum, Events, Exception, Namespace, Property
- Quy tắc Camel: viết thường từ đầu tiên & chữ cái đầu tiên của từ kế tiếp
  - Bổ sung truy cập, Parameter

DATA TYPE

- 2 loại:
  - Kiểu dữ liệu gốc (Primitive Type);
  - Kiểu dữ liệu gốc modifier: sử dụng 1 trong các modifier sau:
    - signed
    - unsigned
    - short
    - long

Kiểu dữ liệu	**Keyword**	**Byte**
Boolean	bool	1
Ký tự	float	1
Số nguyên	int	4
Số thực	float	4
Số thực dạng double	double	8
Kiểu không có giá trị	void	

IDENTIFIER

- Biến: vị trí bộ nhớ được dành riêng để lưu giá trị.
- 3 loại:
  - Biến giá trị;
  - Biến tham chiếu;
  - Biến con trỏ.
- Khai báo & Khởi tạo:
  - int x;
  - x = 5;

  → int x = 5;

IDENTIFIER

- Phân loại theo phạm vi:
  - Biến cục bộ;
  - Biến toàn cục
  - ❖ Toán tử định phạm vi **::**

```
#include <iostream>
using namespace std;
int x = 5;
int main()
{
    int x = 1;
    cout << x << ::x;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int x = 5;
int main()
{
    int y = 1;
    x = y;
    cout << x << y;
    return 0;
}
```

TYPEDEF

- **typedef**: tạo một tên mới cho một kiểu dữ liệu đang tồn tại:

typedef type name;

  - Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    typedef int SoNguyen;
    SoNguyen x = 5;
    cout << x;
    return 0;
}
```

CASTING

- C: **(new_type) expression;**
- C++: vẫn sử dụng như trên.
  - Phép chuyển kiểu mới: **new_type (expression);**

  → Phép chuyển kiểu này có dạng như một hàm số chuyển kiểu đang được gọi.

  - **type_cast <new_type> (expression);**
    - type_cast: const_cast, static_cast, dynamic_cast, reinterpret_cast.

CONSTANT/LITERAL

- Có 2 cách định nghĩa:
  - Sử dụng bộ tiền xử lý **#define**;

    #define identifier value
  - Sử dụng từ khóa **const**;

    const type identifier = value;

    type const identifier = value;

```
#include <iostream>
using namespace std;
#define PI 3.14
int main()
{
    const int x = 5;
    int const y = 6;
    cout << PI << x << y;
    return 0;
}
```

ENUM

- **Kiểu liệt kê**: Là kiểu dữ liệu mà tất cả các phần tử của nó (có ý nghĩa tương đương nhau) có giá trị là hằng số:
  - Được định nghĩa thông qua từ khóa **enum**;
  - Khai báo enum không yêu cầu cấp phát bộ nhớ, chỉ cấp phát bộ nhớ lúc tạo biến kiểu enum;
  - Mặc định các phần tử của enum có kiểu dữ liệu **int**;
  - Enum sẽ được ép kiểu ngầm định sang kiểu int, nhưng kiểu int phải ép kiểu tường mình sang enum;
  - Biến kiểu enum chỉ có thể được gán giá trị là một trong số các hằng đã khai báo bên trong kiểu enum đó, không thể sử dụng hằng của kiểu enum khác;
  - Phạm vi sử dụng của một khai báo enum cũng tượng tự như phạm vi sử dụng khi khai báo biến.

```
#include <iostream>
using namespace std;
int main()
{
    enum Color
    {
        RED = 1,
        BLUE = 2,
        GREEN = 3,
        YELLOW = 4
    };
    Color c = GREEN;
    cout << c;
    return 0;
}
```

OPERATOR

- Toán tử số học: **+, -, *, /, %, ++, --**
- Toán tử quan hệ: **==, !=, >, <, >=, <=**
- Toán tử logic: **&&, ||, !**
- Toán tử so sánh bit: **&, |, ^, ~, <<, >>**
- Toán tử gán: **=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=**
- Toán tử hỗn hợp: **sizeof, ?x:y, ,, &, ***

❖ *Toán tử có thứ tự ưu tiên.*

IOSTREAM.H

- Thư viện:
  - Khai báo tệp tiêu đề:
    - include <iostream.h>
  - Sử dụng câu lệnh nhập, xuất dữ liệu trong C++ cần sử dụng thêm **using namespace std**:

```
#include <iostream>
using namespace std;
int main()
{
    //code
    return 0;
}
```

OUTPUT

- Toán tử xuất:
  - C: ???
  - C++:
    - Cú pháp:

      cout << biểu_thức_1 << … << biểu_thức_n;
    - Trong đó **cout** được định nghĩa trước như một đối tượng biểu diễn cho thiết bị xuất chuẩn của C++ là màn hình;
    - **cout** được sử dụng kết hợp với toán tử chèn << để hiển thị giá trị các biểu thức 1, 2, …, n ra màn hình;
    - Sử dụng "**\n**" hoặc **endl** để xuống dòng mới.

OUTPUT

- Toán tử xuất:
  - Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    int x = 5;
    cout << x << endl;
    cout << "x = " << x << endl;
    return 0;
}
```

INPUT

- Toán tử nhập:
  - C: **???**
  - C++:
    - Cú pháp:

      cin >> biến_1 >> … >> biến_n;
    - Toán tử **cin** được định nghĩa trước như một đối tượng biểu diễn cho thiết bị vào chuẩn của C++ là bàn phím;
    - **cin** được sử dụng kết hợp với toán tử trích >> để nhập dữ liệu từ bàn phím cho các biến 1, 2, …, n.

INPUT

- Xuất dữ liệu:
  - Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    int x;
    cout << "Input x = ";
    cin >> x;
    cout << "x = " << x << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    char x, int y;
    cout << "Input x = ";
    cin >> x;
    y = x;
    cout << x << y << endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    char x, int y;
    cout << "Input y = ";
    cin >> y;
    x = y;
    cout << x << y << endl;
    return 0;
}
```

ĐỊNH DẠNG IN

- Quy định số thực được hiển thị ra màn hình với p chữ số sau dấu chấm thập phân, sử dụng đồng thời các hàm sau:

```
cout << setiosflags(ios::showpoint) << setprecision(p);
```

  - **setiosflags(ios::showpoint)**: bật cờ hiệu showpoint(p).
  - Định dạng trên sẽ có hiệu lực đối với tất cả các toán tử xuất tiếp theo cho đến khi gặp một câu lệnh định dạng mới.

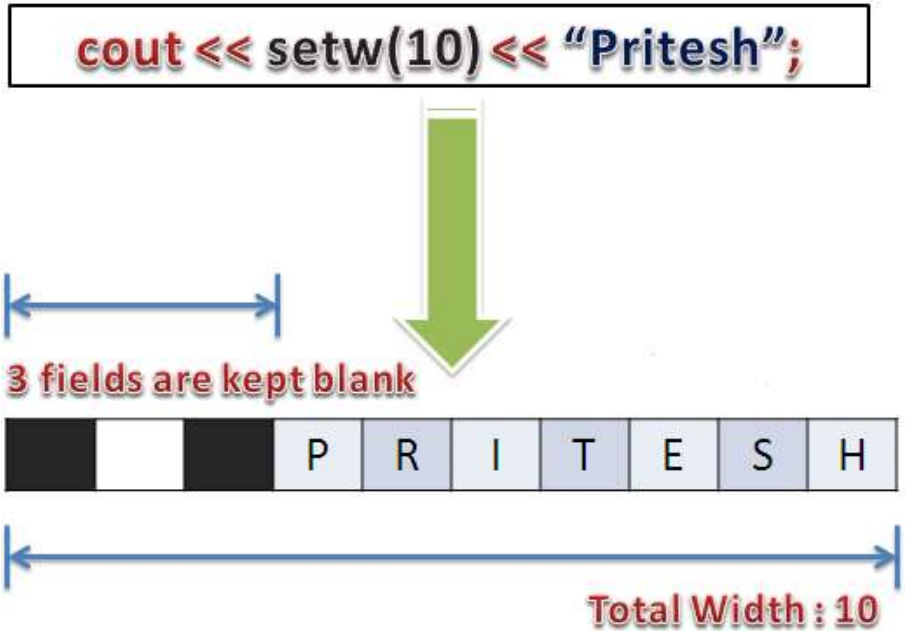
ĐỊNH DẠNG IN

- Ví dụ:

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    double x = 1234.5;
    //x = 1.234; x = 12.34; x = 123.4; x = 1234.5
    cout << setiosflags(ios::showpoint) << setprecision(3);
    cin >> x;
    return 0;
}
```

ĐỊNH DẠNG IN

- Để quy định độ rộng tối thiểu để hiển thị là k vị trí cho giá trị (nguyên, thực, chuỗi) ta dùng hàm: **setw(k);** (**#include <iomanip>**).
- Hàm này cần đặt trong toán tử xuất và nó chỉ có hiệu lực cho một giá trị được in gần nhất. Các giá trị in ra tiếp theo sẽ có độ rộng tối thiểu mặc định là 0.




```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    cout << setw(5) << "KHOA" << "CNTT";
    return 0;
}
```

BIẾN THAM CHIẾU

- Biến tham chiếu là một tên khác (bí danh) cho biến đã định nghĩa:

Kiểu_dữ_liệu &Biến_tham_chiếu = biến_hằng;

  - Biến tham chiếu có đặc điểm là nó được sử dụng làm bí danh cho một biến (kiểu giá trị) nào đó và sử dụng vùng nhớ của biến này;
  - Biến tham chiếu không được cung cấp vùng nhớ (dùng chung địa chỉ vùng nhớ với biến mà nó tham chiếu đến);
  - Trong khai báo biến tham chiếu phải chỉ rõ tham chiếu đến biến nào;
  - Biến tham chiếu có thể tham chiếu đến một phần tử mảng, nhưng không cho phép khai báo mảng tham chiếu.

```
#include <iostream>
using namespace std;
int main()
{
    int x = 5;
    int &y = x;
    y = 1;
    cout << x;
    x++;
    cout << y;
    return 0;
}
```

HẰNG THAM CHIẾU

- Cú pháp:

  const Kiểu_dữ_liệu &Hằng = Biến/Hằng;

- Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    const int a = 1;
    int b = 2;
    const int &x = a; //x++
    const int &y = b; //y++
    const int &z = 3; //z++
    cout << x << y << z;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    int x = 1;
    const int &y = x;
    x++;    //y++;
    cout << x << y;
    return 0;
}
```

HÀM

- Khai báo nguyên mẫu hàm & Định nghĩa hàm;
- Ví dụ:

```
#include <iostream>
using namespace std;
void Sum(int x, int y);
int main()
{
    int m = 1, n = 2;
    Sum(m, n);
    return 0;
}
void Sum(int x, int y)
{
    cout << x + y;
}
```

```
#include <iostream>
using namespace std;
void Sum(int x, int y)
{
    cout << x + y;
}
int main()
{
    int m = 1, n = 2;
    Sum(m, n);
    return 0;
}
```

- Truyền tham trị & tham chiếu (biến tham chiếu hoặc biến con trỏ):

```
#include <iostream>
using namespace std;
void Swap(int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}
int main()
{
    int m = 1, n = 2;
    Swap(m, n);
    cout << m << n;
    return 0;
}
```

```
#include <iostream>
using namespace std;
void Swap(int &x, int &y)
{
    int temp = x;
    x = y;
    y = temp;
}
int main()
{
    int m = 1, n = 2;
    Swap(m, n);
    cout << m << n;
    return 0;
}
```

HÀM VỚI ĐỐI SỐ HẰNG

- Đối số bình thường:

```
#include <iostream>
using namespace std;
void Show(int m)
{
    cout << m++;
}
int main()
{
    int x = 1;
    const int y = 1;
    Show(x);    //1
    Show(y);    //1
    return 0;
}
```

```
#include <iostream>
using namespace std;
void Show(int &m)
{
    cout << m++;
}
int main()
{
    int x = 1;
    const int y = 1;
    Show(x);    //1
    Show(y);    //compile error
    return 0;
}
```

HÀM VỚI ĐỐI SỐ HẰNG

- Đối số hằng: sử dụng khi không muốn thay đổi giá trị đối số truyền vào:

```
#include <iostream>
using namespace std;
void Show(const int m)
{
    cout << m;  //cout << m++;
}
int main()
{
    int x = 1;
    const int y = 1;
    Show(x);    //1
    Show(y);    //1
    return 0;
}
```

HÀM VỚI ĐỐI SỐ HẰNG

- Đối số hằng tham chiếu:

```
#include <iostream>
using namespace std;
void Show(int &m)
{
    m++;
}
int main()
{
    int x = 1;
    const int y = 1;
    Show(x);    //2
    Show(y);    //compile error
    cout << x << y;
    return 0;
}
```

```
#include <iostream>
using namespace std;
void Show(const int &m)
{
    cout << m++;    //compile error
}
void Display(const int &m)
{
    cout << m + 1;
}
int main()
{
    int x = 1;
    Show(x);
    Display(x);
    return 0;
}
```

HÀM VỚI ĐỐI SỐ MẶC ĐỊNH

- Đối số mặc định:
  - Định nghĩa các giá trị tham số mặc định cho các hàm;
  - Các đối số mặc định cần là các đối số cuối cùng tính từ trái qua phải;
  - Nếu chương trình sử dụng khai báo nguyên mẫu hàm thì các đối số mặc định càn được khởi gán trong nguyên mẫu hàm, không được khởi gán lại cho các đối số mặc định trong dòng đầu của định nghĩa hàm;
  - Khi xây dựng hàm, nếu không khai báo nguyên mẫu hàm thì các đối số mặc định được khởi gán trong dòng đầu của định nghĩa hàm;
  - Đối với các hàm có đối số mặc định thì lời gọi hàm cần viết theo quy định: các tham số vắng mặt trong lời gọi hàm tương ứng với các đối số mặc định cuối cùng (tính từ trái sang phải).

HÀM VỚI ĐỐI SỐ MẶC ĐỊNH

- Đối số mặc định:

```
#include <iostream>
using namespace std;
void Show(int x, int y = 1, int z = 2);
int main()
{
    Show();          //compile error
    Show(1);         //112
    Show(1, 2);      //122
    Show(1, 2, 3);   //123
    Show(1, , 1);    //compile error
    return 0;
}
void Show(int x, int y, int z)
{
    cout << x << y << z;
}
```

```
#include <iostream>
using namespace std;
void Show(int x, int y = 1, int z = 2)
{
    cout << x << y << z;
}
int main()
{
    Show();          //compile error
    Show(1);         //112
    Show(1, 2);      //122
    Show(1, 2, 3);   //123
    Show(1, , 1);    //compile error
    return 0;
}
```

HÀM TRẢ VỀ THAM CHIẾU

- Hàm trả về là một tham chiếu:

```
Kiểu   &Tên_hàm(…)
{
        //thân hàm
        return <biến phạm vi toàn cục>;
}
```

  - Biểu thức được trả lại trong câu lệnh **return** phải là biến toàn cục, khi đó mới có thể sử dụng được giá trị của hàm;
  - Nếu trả về tham chiếu đến một biến cục bộ thì biến cục bộ này sẽ bị mất đi khi kết thúc thực hiện hàm. Do vậy tham chiếu của hàm sẽ không còn ý nghĩa nữa. Vì vậy, nếu hàm trả về là tham chiếu đến biến cục bộ thì biến cục bộ này **NÊN** khai báo **static**;
  - Khi giá trị trả về của hàm là tham chiếu, ta có thể gặp các câu lệnh gán hơi khác thường, trong đó vế trái là một lời gọi hàm chứ không phải là tên của một biến.

HÀM TRẢ VỀ THAM CHIẾU

- Ví dụ:

```
#include <iostream>
using namespace std;
int x = 4;
int &Func()
{
    return x;
}
int main()
{
    cout << x;       //4
    cout << Func(); //4
    Func() = 8;
    cout << x;       //8
    system("pause");
    return 0;
}
```

```
#include <iostream>
using namespace std;
int x = 4;
int &Func()
{
    static int x = 5;
    return x;
}
int main()
{
    cout << x;       //4
    cout << Func(); //5
    Func() = 8;
    cout << x;       //4
    system("pause");
    return 0;
}
```

HÀM TRẢ VỀ THAM CHIẾU

- Hàm trả về hằng tham chiếu:

```
#include <iostream>
using namespace std;
int &Func()
{
    static int x = 4;
    return x;
}
int main()
{
    cout << Func();
    cout << Func()++;
    return 0;
}
```

```
#include <iostream>
using namespace std;
const int &Func()
{
    static int x = 4;
    return x;
}
int main()
{
    cout << Func();
    cout << Func()++;    //compile error
    return 0;
}
```

HÀM HẰNG

- Hàm hằng:
  - const int func(int value);
  - int func(int value) const;

```
#include <iostream>
using namespace std;
const int Func()
{
    int x = 4;
    x++;
    return x;
}
int main()
{
    int x = Func();      //x = 4
    int y = Func()++;    //compile error
    return 0;
}
```

HÀM INLINE

- Hàm nội tuyến (inline):
  - Hàm: làm chậm tốc độ thực hiện chương trình vì phải thực hiện một số thao tác có tính thủ tục khi gọi hàm:
    - Cấp phát vùng nhớ cho đối số và biến cục bộ;
    - Truyền dữ liệu của các tham số cho các đối;
    - Giải phóng vùng nhớ trước khi thoát khỏi hàm.
  - C++ cho khả năng khắc phục được nhược điểm trên bằng cách dùng hàm nội tuyến → dùng từ khóa inline trước khai báo nguyên mẫu hàm.
  - ❖Chú ý:
    - Hàm nội tuyến cần có từ khóa inline phải xuất hiện trước các lời gọi hàm;
    - Chỉ nên khai báo là hàm inline khi hàm có nội dung đơn giản;
    - Hàm đệ quy không thể là hàm inline.

HÀM INLINE

- Ví dụ:

```
#include <iostream>
using namespace std;
inline int f(int a, int b);
int main()
{
    int s;
    s = f(5,6);
    cout << s;
    return 0;
}
int f(int a, int b)
{
    return a*b;
}
```

```
#include <iostream>
using namespace std;
int f(int a, int b);
int main()
{
    int s;
    s = f(5,6);
    cout << s;
    return 0;
}
int f(int a, int b)
{
    return a*b;
}
```

CON TRỎ

- Khái niệm:
  - Con trỏ là biến dùng để chứa địa chỉ của biến khác;
  - Cùng kiểu dữ liệu với kiểu dữ liệu của biến mà nó trỏ tới.
- Cú pháp: **<kiểu dữ liệu> *<tên con trỏ>;**

❖ Lưu ý:
  - Có thể viết dấu * ngay sau kiểu dữ liệu;
  - Ví dụ: **int *a;** và **int* a;** là tương đương.
- Thao tác với con trỏ:
  - * là toán tử thâm nhập (dereferencing operator): *p là giá trị nội dung vùng nhớ con trỏ đang trỏ đến;
  - **&** là toán tử địa chỉ (address of operator): &x là địa chỉ của biến x; nếu int *p = &x thì p ↔ x.

CON TRỎ

- Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    int x = 10, y = 20;
    int *p1, *p2;
    p1 = &x; p2 = &y;
    cout << "x = " << x;          //x = 10
    cout << "y = " << y;          //y = 20
    cout << "*p1 = " << *p1;      //*p1 = 10
    cout << "*p2 = " << *p2;      //*p2 = 20
    *p1 = 50; *p2 = 90;
    cout << "x = " << x;          //x = 50
    cout << "y = " << y;          //y = 90
    cout << "*p1 = " << *p1;      //*p1 = 50
    cout << "*p2 = " << *p2;      //*p2 = 90
    *p1 = *p2;
    cout << "x = " << x;          //x = 90
    cout << "y = " << y;          //y = 90
    cout << "*p1 = " << *p1;      //*p1 = 90
    cout << "*p2 = " << *p2;      //*p2 = 90
    return 0;
}
```

CON TRỎ HẰNG & HẰNG CON TRỎ

- Lưu ý:
  - **const int x = 1;** và **int const x = 1;** là như nhau;
  - Nếu **int x = 1;** thì
    - **const int *p = &x; (CON TRỎ HẰNG)**

      Không cho phép thay đổi giá trị vùng nhớ mà con trỏ đang trỏ đến thông qua con trỏ (*p).
    - **int* const p = &x; (HẰNG CON TRỎ)**

      Không cho phép thay đổi vùng nhớ con trỏ đang trỏ tới, nhưng có thể thay đổi giá trị vùng nhớ đó thông qua con trỏ.

CON TRỎ

- Phép gán giữa các con trỏ:
  - Các con trỏ cùng kiểu có thể gán cho nhau thông qua phép gán và lấy địa chỉ con trỏ

    <tên con trỏ 1> = <tên con trỏ 2>;
  - Lưu ý:
    - Bắt buộc phải dùng phép **lấy địa chỉ của biến** do con trỏ trỏ tới mà không được dùng phép **lấy giá trị của biến**;
    - Hai con trỏ phải cùng kiểu dữ liệu (nếu khác kiểu phải sử dụng các phương thức ép kiểu).

CON TRỎ

- Phép gán giữa các con trỏ:
  - Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    int x = 1;
    int *p1, *p2;
    p1 = &x;
    cout << *p1;     //*p1 = 1
    *p1 += 2;
    cout << x;       //x = 3
    p2 = p1;
    *p2 += 3;
    cout << x;       //x = 6
    return 0;
}
```

CON TRỎ

- Sử dụng **typedef**:

```
#include <iostream>
using namespace std;
typedef int P;
typedef int* Q;
int main()
{
    int x = 1, y = 2;
    P *p1, *p2;
    p1 = &x; p2 = &y;
    cout << *p1 << *p2; //12
    Q p3, p4;
    p3 = &x; p4 = &y;
    cout << *p3 << *p4; //12
    P* p5, p6;
    p5 = &x;
    p6 = &y;    //compile error
    return 0;
}
```

CONST_CAST

- **const_cast**: thêm hoặc loại bỏ const khỏi một biến (là cách duy nhất để thay đổi tính hằng của biến).
- Ví dụ:

```
#include <iostream>
using namespace std;
void Show(char *str)
{
    cout << str;
}
int main()
{
    const char *str = "DUT";
    Show(str);  //compile error
    Show(const_cast<char*>(str));
    return 0;
}
```

MỘT SỐ LƯU Ý

- Đối số con trỏ;
- Hàm trả về là tham chiếu.

```
#include <iostream>
using namespace std;
void Swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}
int main()
{
    int m = 1, n = 2;
    Swap(&m, &n);
    cout << m << n;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int &Func()
{
    int x = 5;
    return x;
}
int main()
{
    int *p = &Func();
    //cout << *p << *p;
    cout << *p;
    cout << *p;
    return 0;
}
```

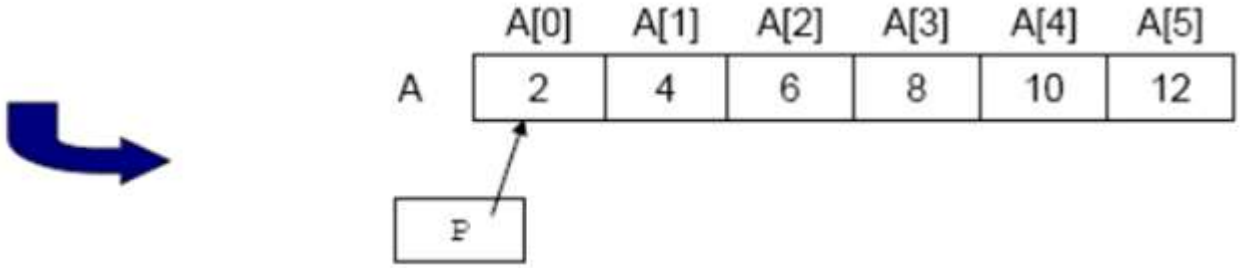
MẢNG

- Khai báo & khởi tạo mảng:
  - int A[5];
  - int A[5] = {1, 2, 3, 4, 5};
  - int A[] = {1, 2, 3, 4};
  - int A[2][3];
  - int A[2][3] = {{1, 2, 3}, {4, 5, 6}};
  - int A[2][3] = {1, 2, 3, 4, 5, 6};
  - int A[][3] = {{1, 2, 3}, {4, 5, 6}};
    - Không được bỏ trống cột với mảng 2 chiều.
- Truy cập phần tử trong mảng:
  - A[i] – giá trị phần tử thứ i; &A[i] – địa chỉ phần tử thứ i;
  - A[i][j] – giá trị phần tử hàng i, cột j;
  - &A[i] – địa chỉ hàng i;
  - &A[i][j] – địa chỉ phần tử hàng i, cột j.

CON TRỎ & MẢNG

- Quan hệ giữa con trỏ và mảng 1 chiều:
  - Tên mảng được coi như một con trỏ tới phần tử đầu tiên của mảng:

```
int A[6] = {2,4,6,8,10,12};
int *P;
P = A; // P points to A
```



  - Do tên mảng và con trỏ là tương đương, ta có thể dùng P như tên mảng. Ví dụ:

P[3] = 7; tương đương với **A[3] = 7;**

CON TRỎ & MẢNG

- Phép toán trên con trỏ và mảng 1 chiều:
  - Khi con trỏ trỏ đến mảng, thì các phép toán tăng hay giảm trên con trỏ sẽ tương ứng với phép dịch chuyển trên mảng;
  - Lưu ý: nếu **int *p = A;** thì **p++** và ***p++** là khác nhau
    - **p++** thao tác trên con trỏ: đưa con trỏ pa đến địa chỉ phần tử tiếp theo;
    - ***p++** là phép toán trên giá trị, tăng giá trị phần tử hiện tại của mảng.

CON TRỎ & MẢNG

- Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    int A[] = {1, 2, 3, 4, 5};
    int *p = A;
    p += 2;
    cout << *p; //*p = A[2] = 3
    p--;
    cout << *p; //*p = A[1] = 2
    *p++;
    cout << *p; //*p = A[1] = 3
    return 0;
}
```

CIN.GET()

- **Lưu ý:**
  - cin.get(a, n); - nhập một chuỗi không quá n ký tự và lưu vào mảng một chiều a (kiểu char);
    - Toán tử nhập **cin>>** sẽ để lại ký tự chuyển dòng '**\n**' trong bộ đệm → làm trôi phương thức **cin.get**;
    - Để khắc phục tình trạng trên cần dùng phương thức **cin.ignore(1)** để bỏ qua một ký tự chuyển dòng.

CON TRỎ & MẢNG

- Con trỏ & mảng 2 chiều
  - Địa chỉ ma trận A:

    A = A[0] = *(A+0) = &A[0][0];
  - Địa chỉ của hàng thứ nhất:

    A[1] = *(A+1) = &A[1][0];
  - Địa chỉ của hàng thứ i:

    A[i] = *(A+i) = &A[i][0];
  - Địa chỉ phần tử:

    &A[i][j] = (*(A+i)) + j;
  - Giá trị phần tử:

    A[i][j] = *((*(A+i)) + j);

→ int A[3][3]; tương đương int (*A)[3];.

CON TRỎ & MẢNG

- Con trỏ tới con trỏ:
  - **int A[3][3];** tương đương **int (*A)[3];**
  - **int A[3];** tương đương **int *A;**
  - →Mảng 2 chiều có thể thay thế bằng một mảng các con trỏ tương đương một con trỏ trỏ đến con trỏ.
  - Ví dụ:

```
int A[3][3];
int (*A)[3];
int **A;
```

CON TRỎ & MẢNG

- Con trỏ & mảng 2 chiều
  - Bài tập: Chương trình sau gồm các hàm:
    - Nhập một ma trận thực cấp mxn
    - In một ma trận thực dưới dạng bảng
    - Tìm phần tử lớn nhất và phần tử nhỏ nhất của dãy số thực;

  Viết chương trình sẽ nhập một ma trận, in ma trận vừa nhập và in các phần tử lớn nhất và nhỏ nhất trên mỗi hàng của ma trận.

FUNCTION POINTER

- Mỗi hàm đều có 1 địa chỉ xác định trên bộ nhớ → có thể sử dụng con trỏ để trỏ đến địa chỉ của hàm.

```
#include <iostream>
using namespace std;
int Show(int x)
{
    return x;
}
int main()
{
    int n = 5;
    cout << Show(n);     //5
    cout << Show;    //Address of function Show
    return 0;
}
```

- **Với compiler của Visual Studio (Window) thì kết quả đúng, với compiler của Java thì kết quả bằng 1.**

FUNCTION POINTER

- Khai báo:

  <kiểu dữ liệu trả về> (*<tên hàm>) ([<danh sách tham số>])

- Lưu ý:
  - Dấu "()" bao bọc tên hàm để thông báo đó là con trỏ hàm;
  - Nếu không có dấu "()" thì trình biên dịch sẽ hiểu ta đang khai báo một hàm có giá trị trả về là một con trỏ.
    - Ví dụ:

    ```
    int (*Cal) (int a, int b)      //Khai báo con trỏ hàm
    int *Cal (int a, int b)        //Khai báo hàm trả về kiểu con trỏ
    ```

  - Đối số mặc định không áp dụng cho con trỏ hàm, vì đối số mặc định được cấp phát khi biên dịch, còn con trỏ hàm được sử dụng lúc thực thi.

FUNCTION POINTER

- Ví dụ:

```
#include <iostream>
using namespace std;
void Swap(int &x, int &y)
{
    int temp = x;
    x = y;
    y = temp;
}
int main()
{
    int m = 5, n = 10;
    void(*pSwap) (int &, int &) = Swap;
    (*pSwap)(m, n); //m = 10, n = 5
    return 0;
}
```

FUNCTION POINTER

- Sử dụng con trỏ hàm làm đối số:
  - Sử dụng khi cần gọi 1 hàm như là tham số của 1 hàm khác;
  - Ví dụ: int (*Cal) (int a, int b);
    - Thì có thể gọi các hàm có hai tham số kiểu int và trả về cũng kiểu int:

      int add(int a, int b);

      int sub(int a, int b);
    - Nhưng không được gọi các hàm khác kiểu tham số hoặc kiểu trả về:

      int add(float a, int b);

      int add(int a);

      char* sub(char* a, char* b);

FUNCTION POINTER

- Sử dụng con trỏ hàm làm đối số:
  - Ví dụ: sắp xếp dữ liệu trong mảng số nguyên theo thứ tự tăng dần

```
#include <iostream>
#include <iomanip>
using namespace std;
void Show(int *p, int length)
{

    for (int i = 0; i < length; i++)
        cout << setw(3) << *(p + i);
}
void Swap(int &x, int &y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

```
void SelectionSort(int *p, int length)
{
    for (int i = 0; i < length - 1; i++)
        for (int j = i + 1; j < length; j++)
            if (*(p + i) > *(p + j))
                Swap(*(p + i), *(p + j));
}
int main()
{
    int A[] = {1, 4, 2, 3, 6, 5, 8, 9, 7};
    int l = sizeof(A) / sizeof(int);
    SelectionSort(A, l);
    Show(A, l);
    return 0;
}
```

  - Nhưng nếu muốn sắp xếp theo thứ tự giảm dần ???

FUNCTION POINTER

- Sử dụng con trỏ hàm làm đối số:
  - Thêm 2 hàm so sánh:

```
bool ascending(int left, int right)
{
    return left > right;
}
bool descending(int left, int right)
{
    return left < right;
}
```

  - ascending: sắp xếp tăng dần
  - Descending: sắp xếp giảm dần

FUNCTION POINTER

- Sử dụng con trỏ hàm làm đối số:

```
void SelectionSort(int *p, int length, bool CompFunc(int, int))
{
    for (int i = 0; i < length - 1; i++)
        for (int j = i + 1; j < length; j++)
            if (CompFunc(*(p + i), *(p + j)))
                Swap(*(p + i), *(p + j));
}
int main()
{
    int A[] = {1, 4, 2, 3, 6, 5, 8, 9, 7};
    int l = sizeof(A) / sizeof(int);
    SelectionSort(A, l, ascending);
    SelectionSort(A, l, descending);
    return 0;
}
```

- Lưu ý: Con trỏ hàm là đối số mặc định

```
void SelectionSort(int *p, int length,
    bool CompFunc(int, int) = ascending)
```

STATIC & DYNAMIC ALLOWCATION

- **Cấp phát tĩnh (static allocation)**: biến được cấp phát vùng nhớ khi biên dịch;
- **Cấp phát động (dynamic allocation)**: biến được cấp phát vùng nhớ lúc thực thi:
  - Sử dụng từ khóa **new**;
  - Dùng biến con trỏ (**p**) để lưu trữ địa chỉ vùng nhớ:
    - Cấp phát bộ nhớ 1 biến: **p = new type;**
    - Cấp phát bộ nhớ 1 mảng: **p = new type[n];**

STATIC & DYNAMIC ALLOWCATION

- Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    //Cấp phát tĩnh
    int x = 1; int *p1 = &x;
    //Cấp phát động
    int *p2 = new int; *p2 = 2;
    cout << x;      //1
    cout << *p1;    //1
    cout << *p2;    //2
    p1 = p2;
    p2 = new int; *p2 = 3;
    cout << *p1;    //2
    cout << *p2;    //3
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    //Cấp phát tĩnh
    int A[] = {1, 2, 3, 4}; int *p1 = A;
    //Cấp phát động
    int m = 5;
    int *p2 = new int[m];
    for (int i = 0; i < m; i++)
        *(p2 + i) = i++;
    return 0;
}
```

STATIC & DYNAMIC ALLOWCATION

- Kiểm tra vùng nhớ cấp phát có thành công hay không?
  - *Thành công*: con trỏ chứa địa chỉ đầu vùng nhớ được cấp phát; *Không thành công*: p = NULL.
  - Kiểm tra vùng nhớ cấp phát có thành công hay không bằng con trỏ hàm: new_handler (new.h)
  - Ví dụ:

```
#include <iostream>
using namespace std;
int main()
{
    int *p = new int;
    if (p == NULL)
    {
        cout << "Error";
        exit(0);
    }
    return 0;
}
```

STATIC & DYNAMIC ALLOWCATION

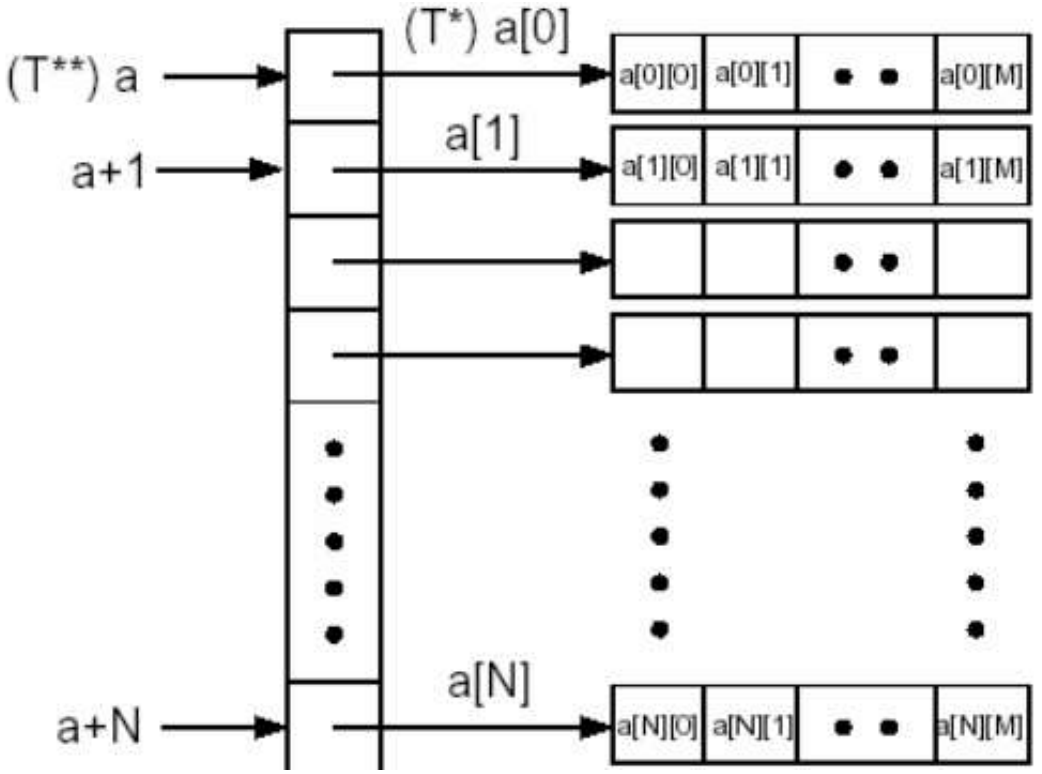
- Giải phóng bộ nhớ trong C++:

```
#include <iostream>
using namespace std;
int main()
{
    int *p1 = new int;
    *p1 = 1;
    delete p1;
    int n = 3;
    int *p2 = new int[n];
    for (int i = 0; i < n; i++)
        *(p2 + i) = i++;
    delete[] p2;
    return 0;
}
```

STATIC & DYNAMIC ALLOWCATION

- Cấp phát động mảng đa chiều:
  - Cấp phát động mảng hai chiều (N+1)(M+1) gồm các đối tượng IQ:

```
IQ **a = new (IQ*) [N+1];
for (int i=0; i<N+1; i++)
    a[i] = new IQ[M+1];
```



STATIC & DYNAMIC ALLOWCATION

- Huỷ mảng động bất hợp lệ:

```
#include <iostream>
int main ()
{
   int* A = new int[6];
   // dynamically allocate array
   A[0] = 0; A[1] = 1; A[2] = 2;
   A[3] = 3; A[4] = 4; A[5] = 5;
   int *p = A + 2;
   cout << "A[1] = " << A[1] << endl;
   delete [] p; // illegal!!!
   // results depend on particular compiler
   cout << "A[1] = " << A[1] << endl;
}
```

P không trỏ tới đầu mảng A

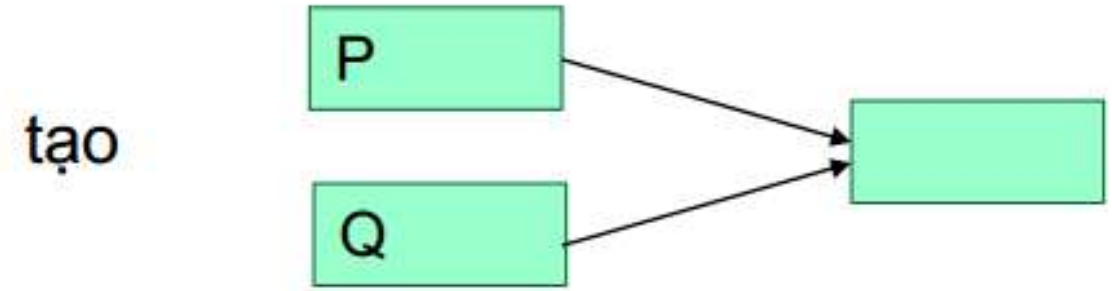
Huỷ không hợp lệ

Kết quả phụ thuộc trình biên dịch

DANGLING POINTER

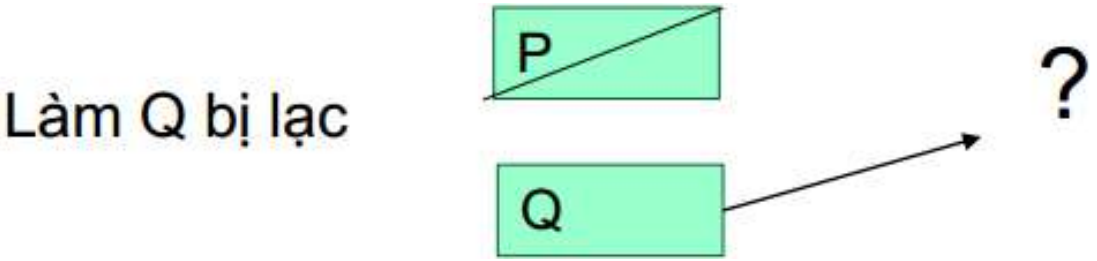
- Con trỏ lạc: khi delete P, ta cần chú ý không xoá vùng bộ nhớ mà một con trỏ Q khác đang trỏ tới:

```
int *P;
int *Q;
P = new int;
Q = P;
```



- Sau đó:

```
delete P;
P = NULL;
```

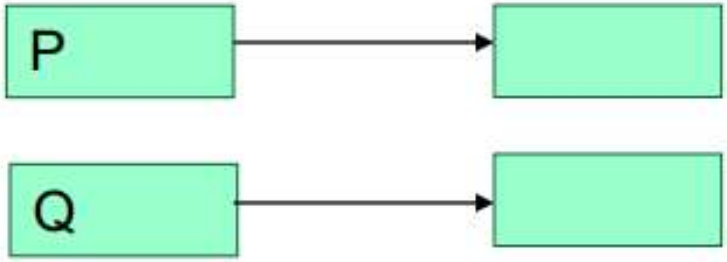


RÒ RỈ BỘ NHỚ

- Rò rỉ bộ nhớ:
  - Một vấn đề liên quan: *mất* mọi con trỏ đến một vùng bộ nhớ được cấp phát. Khi đó, vùng bộ nhớ đó bị mất dấu, không thể trả lại cho heap được:

```
int *P;
int *Q;
P = new int;
Q = new int;
```

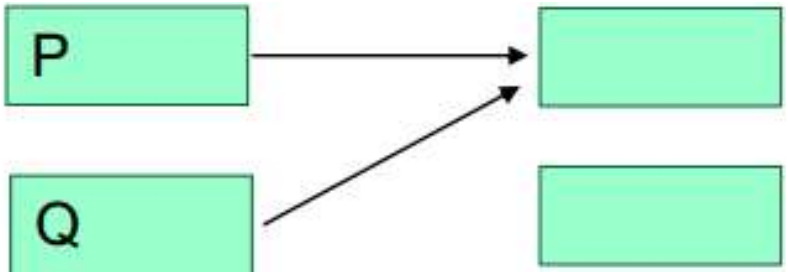
tạo



  - Sau đó:

```
Q = P;
```

Làm mất vùng nhớ đã từng được Q trỏ tới

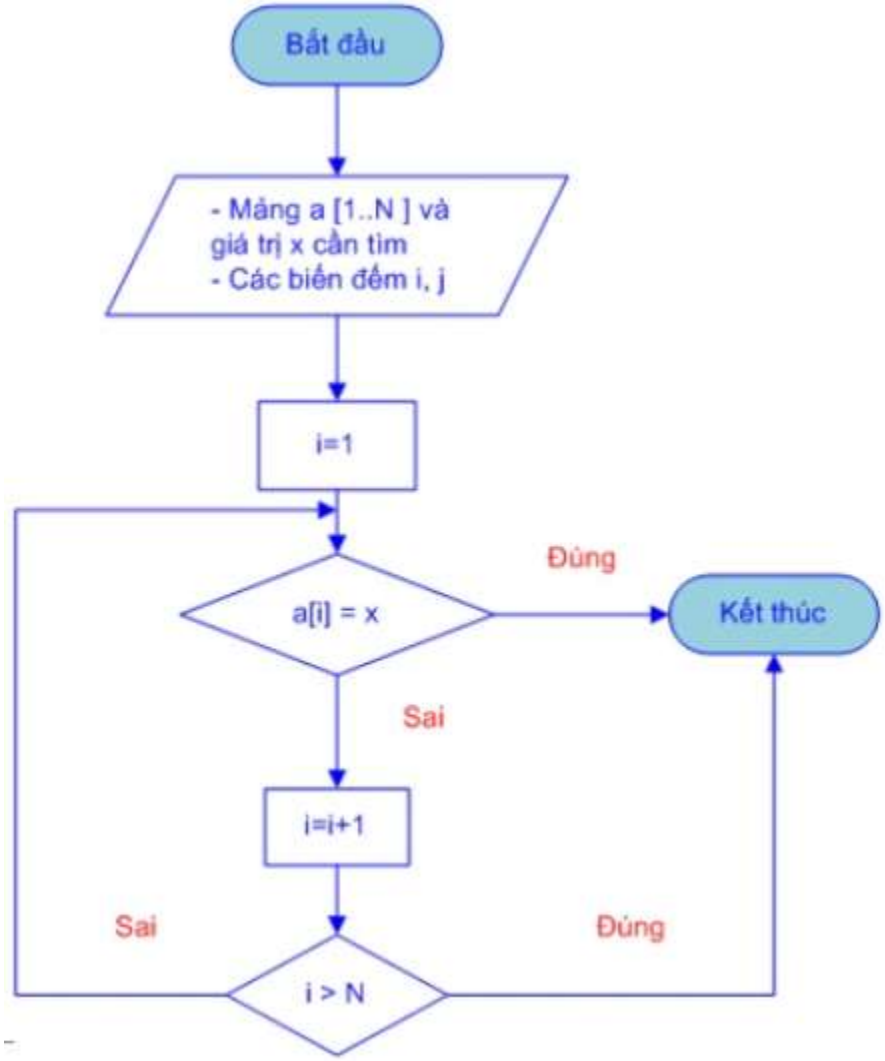


LINEAR SREACH

- Thường sử dụng với các mảng chưa được sắp xếp.
- Ý tưởng: tiến hành so sánh số x với các phần tử trong mảng cho đến khi gặp được phần tử có giá trị cần tìm.
- Input:
  - Mảng A[n] và giá trị cần tìm x
- Output:
  - True: nếu tìm thấy;
  - False: nếu không tìm thấy.

LINEAR SREACH

- Lưu đồ giải thuật:



LINEAR SREACH

- Ví dụ:

```
bool LinearSearch(int *p, int length, int x)
{
    for (int i = 0; i < length; i++)
        if (*(p + i) == length)
            return true;
    return false;
}
```

BINARY SREACH

- Thường dùng cho mảng đã sắp xếp thứ tự;
- Ý tưởng:
  - Tìm kiếm kiểu tra "**từ điển**";
  - Tìm cách giới hạn phạm vi tìm kiếm sau mỗi lần so sánh x với một phần tử trong mảng đã được sắp xếp;
  - Tại mỗi bước, so sánh x với phần tử nằm ở vị trí giữa của mảng tìm kiếm hiện hành:
    - Nếu x nhỏ hơn thì sẽ tìm kiếm ở nửa mảng trước;
    - Ngược lại, tìm kiếm ở nửa mảng sau.

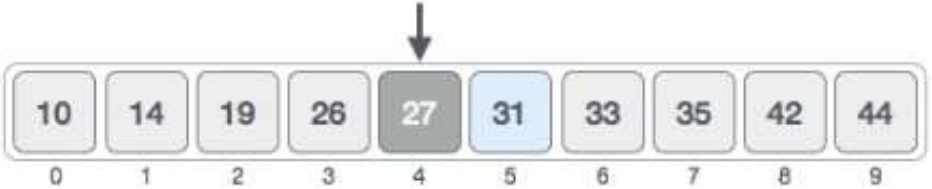
BINARY SREACH

Giả sử chúng ta cần tìm vị trí của giá trị 31 trong một mảng bao gồm các giá trị như hình dưới đây bởi sử dụng Binary Search:

Đầu tiên, chúng ta chia mảng thành hai nửa theo phép toán sau:

```
chỉ-mục-giữa = bắt-đầu + (cuối + bắt-đầu)/ 2
```

Với ví dụ trên là 0 + (9 – 0)/ 2 = 4 (giá trị là 4.5). Do đó 4 là chỉ mục giữa của mảng.



Bây giờ chúng ta so sánh giá trị phần tử giữa với phần tử cần tìm. Giá trị phần tử giữa là 27 và phần tử cần tìm là 31, do đó là không kết nối. Bởi vì giá trị cần tìm là lớn hơn nên phần tử cần tìm sẽ nằm ở mảng con bên phải phần tử giữa.

Chúng ta thay đổi giá trị bắt-đầu thành chỉ-mục-giữa + 1 và lại tiếp tục tìm kiếm giá trị chỉ-mục-giữa.

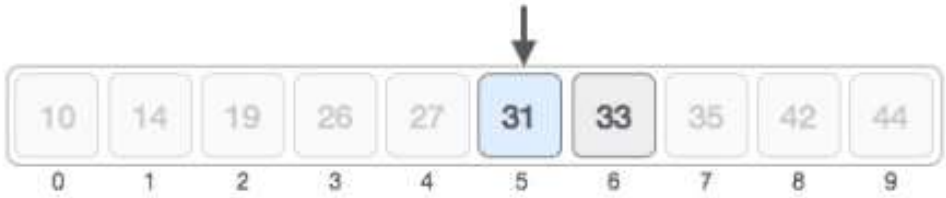
```
bắt-đầu = chỉ-mục-giữa + 1
chỉ-mục-giữa = bắt-đầu + (cuối + bắt-đầu)/ 2
```

BINARY SREACH

Bây giờ chỉ mục giữa của chúng ta là 7. Chúng ta so sánh giá trị tại chỉ mục này với giá trị cần tìm.

Giá trị tại chỉ mục 7 là không kết nối, và ngoài ra giá trị cần tìm là nhỏ hơn giá trị tại chỉ mục 7 do đó chúng ta cần tìm trong mảng con bên trái của chỉ mục giữa này.

Tiếp tục tìm chỉ-mục-giữa lần nữa. Lần này nó có giá trị là 5.

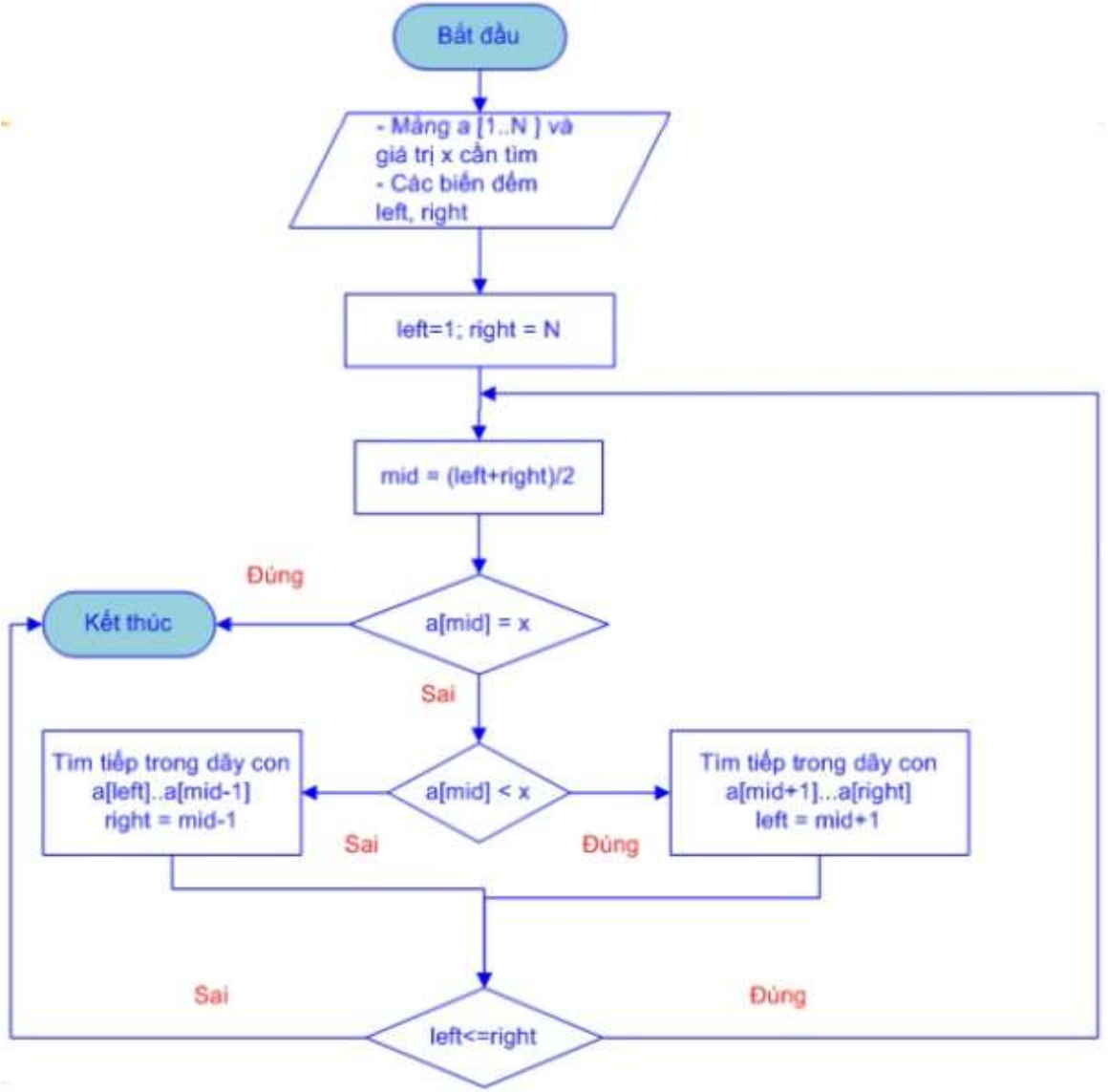


So sánh giá trị tại chỉ mục 5 với giá trị cần tìm và thấy rằng nó kết nối.

Do đó chúng ta kết luận rằng giá trị cần tìm 31 được lưu giữ tại vị trí chỉ mục 5.

BINARY SREACH

- Lưu đồ thuật toán:



BINARY SREACH

- Ví dụ:

```
bool BinarySearch(int *p, int length, int x)
{
    int left = 0, right = length - 1;
    while (left <= right)
    {
        int k = (left + right) / 2;
        if (*(p + k) == x)
            return true;
        if (*(p + k) > x)
            right = k - 1;
        else
            left = k + 1;
    }
    return false;
}
```

INTERPOLATION SREACH

- Biến thể của Binary Search, sử dụng với mảng đã được sắp xếp;
- Ý tưởng:
  - Step 1:
    - Binary Search: pos = left + (right – left)/2;
    - Cải tiến: pos = left + **(X- T[left]) * (right – left)/(T[right] – T[left])**
  - Step 2:
    - Kiểm A[pos] nếu bằng X thì pos là vị trí cần tìm;
    - Nếu nhỏ hơn X thì ta tăng left lên một đơn vị và tiếp tục thực hiện lại bước 1;
    - Nếu lớn hơn X thì ta giảm right một đơn vị và tiếp tục thực hiện lại bước 1.

INTERPOLATION SREACH

Tìm kiếm nội suy tìm kiếm một phần tử cụ thể bằng việc tính toán vị trí dò (Probe Position). Ban đầu thì vị trí dò là vị trí của phần tử nằm ở giữa nhất của tập dữ liệu.

Nếu tìm thấy kết nối thì chỉ mục của phần tử được trả về. Để chia danh sách thành hai phần, chúng ta sử dụng phương thức sau:

```
mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) * (X - A[Lo])

Trong đó:
   A    = danh sách
   Lo   = chỉ mục thấp nhất của danh sách
   Hi   = chỉ mục cao nhất của danh sách
   A[n] = giá trị được lưu giữ tại chỉ mục n trong danh sách
```

Nếu phần tử cần tìm có giá trị lớn hơn phần tử ở giữa thì phần tử cần tìm sẽ ở mảng con bên phải phần tử ở giữa và chúng ta lại tiếp tục tính vị trí dò; nếu không phần tử cần tìm sẽ ở mảng con bên trái phần tử ở giữa. Tiến trình này tiến tụp diễn ra trên các mảng con cho tới khi kích cỡ của mảng con giảm về 0.

INTERPOLATION SREACH

- Ví dụ:

```
bool InterPolationSearch(int *p, int length, int x)
{
    int left = 0;
    int right = length - 1;
    while (left <= right && x >= *(p + left) && x <= *(p + right))
    {
        int val1 = (x - *(p + left)) / (*(p + right) - *(p + left));
        int val2 = right - left;
        int pos = left + val1 * val2;
        if (*(p + pos) == x)
            return true;
        if (*(p + pos) < x)
            left = pos + 1;
        else
            right = pos - 1;
    }
    return false;
}
```

BUBBLE SORT

- Sắp xếp nổi bọt;
- Ý tưởng: xuất phát từ đầu mảng, so sánh 2 phần tử cạnh nhau để đưa phần tử nhỏ hơn lên trước; sau đó lại xét cặp tiếp theo cho đến khi tiến về đầu mảng. Nhờ vậy, ở lần xử lý thứ i sẽ tìm được phần tử ở vị trí đầu mảng là i.
  - Step 1: i = 0
  - Step 2: j = n – 1
    - Trong khi j > i thực hiện: nếu a[j] < a[j – 1]: hoán vị a[j] & a[j – 1], j++;
  - Step 3: i++
    - Nếu i > n + 1 thì dừng, ngược lại, lặp lại step 2.

BUBBLE SORT

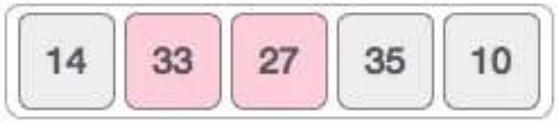
Giả sử chúng ta có một mảng không có thứ tự gồm các phần tử như dưới đây. Bây giờ chúng ta sử dụng giải thuật sắp xếp nổi bọt để sắp xếp mảng này.

Giải thuật sắp xếp nổi bọt bắt đầu với hai phần tử đầu tiên, so sánh chúng để kiểm tra xem phần tử nào lớn hơn.

Trong trường hợp này, 33 lớn hơn 14, do đó hai phần tử này đã theo thứ tự. Tiếp đó chúng ta so sánh 33 và 27.

BUBBLE SORT

Chúng ta thấy rằng 33 lớn hơn 27, do đó hai giá trị này cần được tráo đổi thứ tự.



Mảng mới thu được sẽ như sau:

Tiếp đó chúng ta so sánh 33 và 35. Hai giá trị này đã theo thứ tự.

Sau đó chúng ta so sánh hai giá trị kế tiếp là 35 và 10.

Vì 10 nhỏ hơn 35 nên hai giá trị này chưa theo thứ tự.

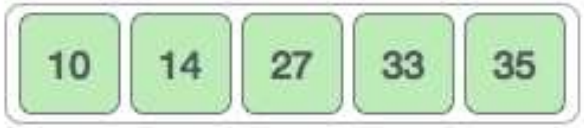
BUBBLE SORT

Tráo đổi thứ tự hai giá trị. Chúng ta đã tiến tới cuối mảng. Vậy là sau một vòng lặp, mảng sẽ trông như sau:

Để đơn giản, tiếp theo mình sẽ hiển thị hình ảnh của mảng sau từng vòng lặp. Sau lần lặp thứ hai, mảng sẽ trông giống như:

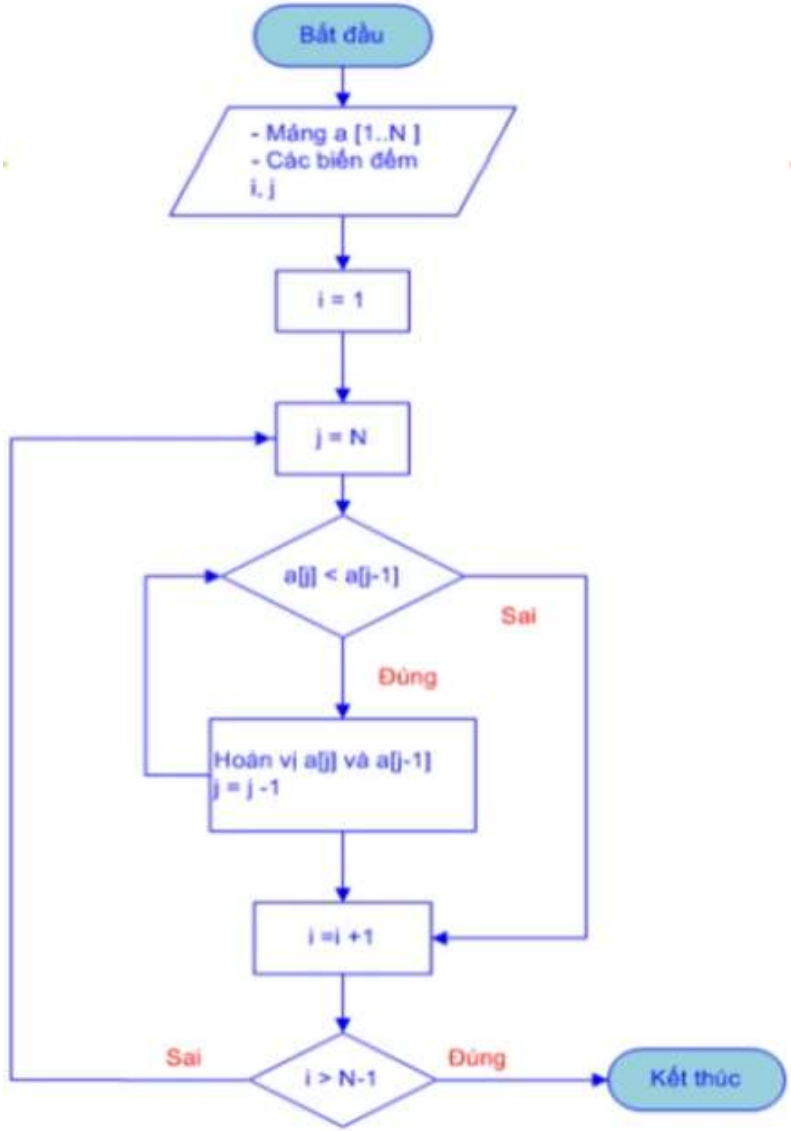
Sau mỗi vòng lặp, ít nhất một giá trị sẽ di chuyển tới vị trí cuối. Sau vòng lặp thứ 3, mảng sẽ trông giống như:

Và khi không cần tráo đổi thứ tự phần tử nào nữa, giải thuật sắp xếp nổi bọt thấy rằng mảng đã được sắp xếp xong.



BUBBLE SORT

- Lưu đồ giải thuật:



BUBBLE SORT

- Ví dụ:

	3	10	4	6	2	6	15	3	9	7
Bước 1	2	3	10	4	6	3	6	15	7	9
Bước 2		3	3	10	4	6	6	7	15	9
Bước 3			3	4	10	6	6	7	9	15
Bước 4				4	6	10	6	7	9	15
Bước 5					6	6	10	7	9	15
Bước 6						6	7	10	9	15
Bước 7							7	9	10	15
Bước 8								9	10	15
Bước 9									10	15
Kết quả	2	3	3	4	6	6	7	9	10	15

BUBBLE SORT

- Ví dụ:

```
void BubbleSort(int *p, int length)
{
    for (int i = 0; i < length - 1; i++)
        for (int j = length - 1; j > i; j--)
            if (*(p + j) < *(p + j - 1))
                Swap(*(p + j - 1), *(p + j));
}
```

SELECTION SORT

- Sắp xếp chọn;
- Ý tưởng:
  - Chọn phần tử nhỏ nhất trong n phần tử ban đầu của mảng, đưa phần tử này về vị trí đầu tiên của mảng; sau đó loại nó ra khỏi danh sách sắp xếp;
  - Mảng hiện hành còn n – 1 phần tử của mảng ban đầu, bắt đầu từ vị trí thứ 2; lặp lại quá trình trên cho mảng hiện hành đến khi mảng hiện hành còn 1 phần tử.

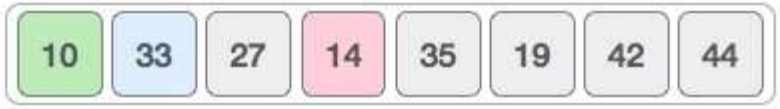
SELECTION SORT

Từ vị trí đầu tiên trong danh sách đã được sắp xếp, toàn bộ danh sách được duyệt một cách liên tục. Vị trí đầu tiên có giá trị 14, chúng ta tìm toàn bộ danh sách và thấy rằng 10 là giá trị nhỏ nhất.

Do đó, chúng ta thay thế 14 với 10. Sau một vòng lặp, giá trị 10 thay thế cho giá trị 14 tại vị trí đầu tiên trong danh sách đã được sắp xếp. Chúng ta tráo đổi hai giá trị này.

Tại vị trí thứ hai, giá trị 33, chúng ta tiếp tục quét phần còn lại của danh sách theo thứ tự từng phần tử.

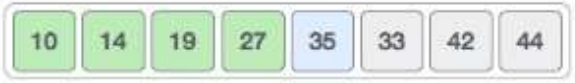
Chúng ta thấy rằng 14 là giá trị nhỏ nhất thứ hai trong danh sách và nó nên xuất hiện ở vị trí thứ hai. Chúng ta tráo đổi hai giá trị này.

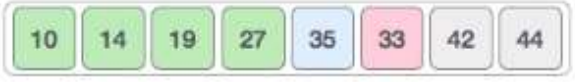


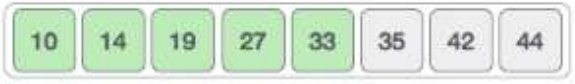
SELECTION SORT

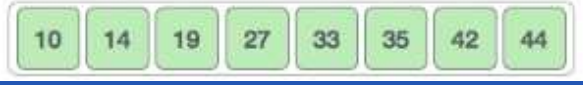
Sau hai vòng lặp, hai giá trị nhỏ nhất đã được đặt tại phần đầu của danh sách đã được sắp xếp.

Tiến trình tương tự sẽ được áp dụng cho phần còn lại của danh sách. Các hình dưới minh họa cho các tiến trình này.



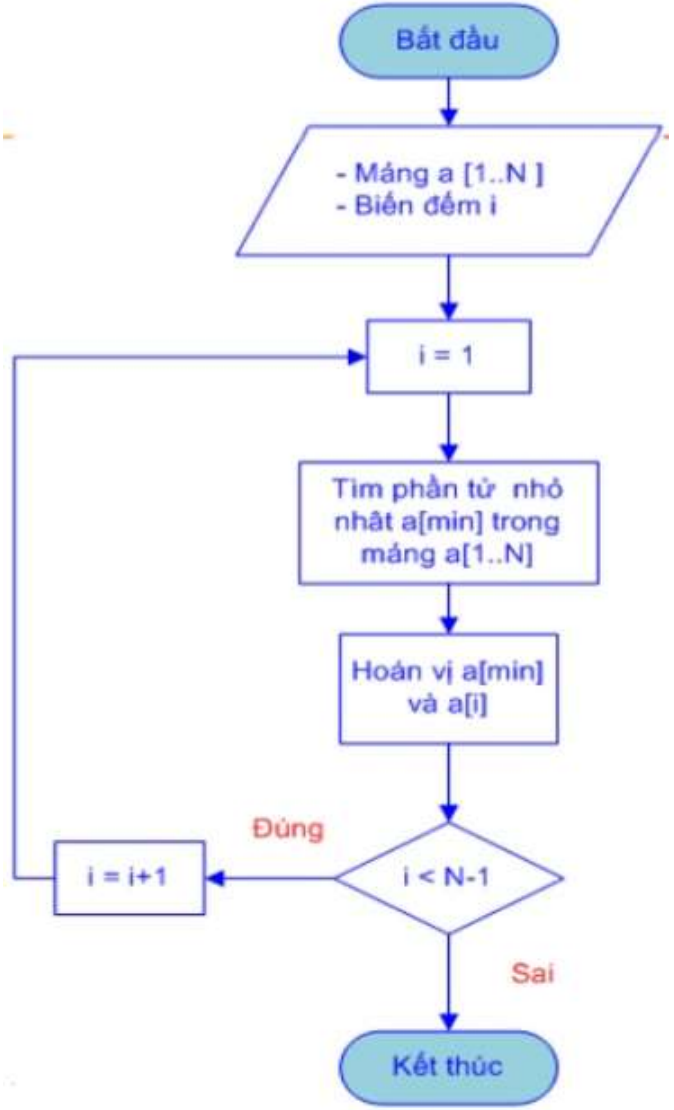






SELECTION SORT

- Lưu đồ giải thuật:



SELECTION SORT

- Ví dụ:

```
void SelectionSort(int *p, int length)
{
    for (int i = 0; i < length - 1; i++)
        for (int j = i + 1; j < length; j++)
            if (*(p + i) > *(p + j))
                Swap(*(p + i), *(p + j));
}
```

INSERTION SORT

- Sắp xếp chèn;
- Ý tưởng:

Ví dụ chúng ta có một mảng gồm các phần tử không có thứ tự:

Giải thuật sắp xếp chèn so sánh hai phần tử đầu tiên:

Giải thuật tìm ra rằng cả 14 và 33 đều đã trong thứ tự tăng dần. Bây giờ, 14 là trong danh sách con đã qua sắp xếp.

Giải thuật sắp xếp chèn tiếp tục di chuyển tới phần tử kế tiếp và so sánh 33 và 27.

INSERTION SORT

Giải thuật sắp xếp chèn tráo đổi vị trí của 33 và 27. Đồng thời cũng kiểm tra tất cả phần tử trong danh sách con đã sắp xếp. Tại đây, chúng ta thấy rằng trong danh sách con này chỉ có một phần tử 14 và 27 là lớn hơn 14. Do vậy danh sách con vẫn giữ nguyên sau khi đã tráo đổi.

Bây giờ trong danh sách con chúng ta có hai giá trị 14 và 27. Tiếp tục so sánh 33 với 10.

Hai giá trị này không theo thứ tự.

Vì thế chúng ta tráo đổi chúng.

Việc tráo đổi dẫn đến 27 và 10 không theo thứ tự.

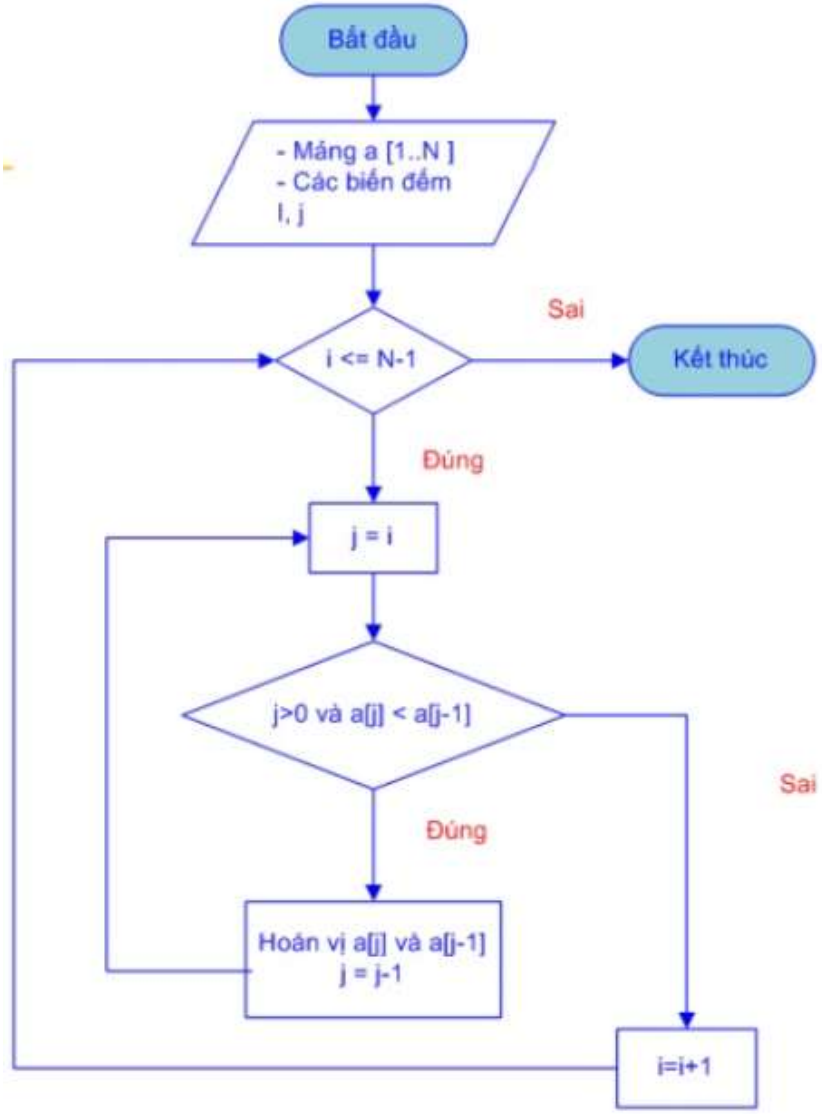
Vì thế chúng ta cũng tráo đổi chúng.

Chúng ta lại thấy rằng 14 và 10 không theo thứ tự.

Và chúng ta tiếp tục tráo đổi hai số này. Cuối cùng, sau vòng lặp thứ 3 chúng ta có 4 phần tử.

INSERTION SORT

- Lưu đồ giải thuật:



INSERTION SORT

- Ví dụ:

	3	7	22	3	1	5	8	4	3	9
Bước 0	3									
Bước 1	3	7								
Bước 2	3	7	22							
Bước 3	3	3	7	22						
Bước 4	1	3	3	7	22					
Bước 5	1	3	3	5	7	22				
Bước 6	1	3	3	5	7	8	22			
Bước 7	1	3	3	4	5	7	8	22		
Bước 8	1	3	3	3	4	5	7	8	22	
Bước 9	1	3	3	3	4	5	7	8	9	22

INSERTION SORT

- Ví dụ:

```
void InsertionSort(int *p, int length)
{
    for (int i = 0; i < length; i++)
        for (int j = i; j > 0; j--)
            if (*(p + j) < *(p + j - 1))
                Swap(*(p + j), *(p + j - 1));
}
```

MEGRE SORT

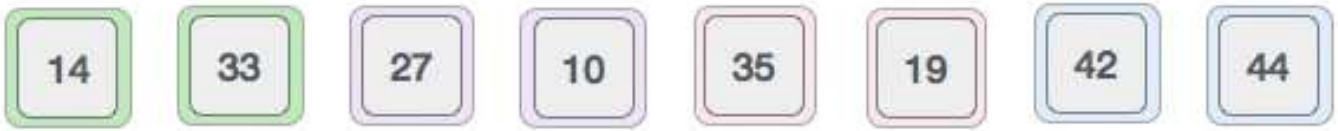
- Sắp xếp trộn;
- Ý tưởng:
  - Chia mảng lớn thành những mảng con nhỏ hơn;
  - Tiếp tục chia cho đến khi mảng con nhỏ nhất là 1 phần tử;
  - Tiến hành so sánh 2 mảng con có cùng mảng cơ sở (vừa so sánh, vừa sắp xếp & ghép) cho đến khi gộp thành 1 mảng duy nhất

MEGRE SORT

Đầu tiên, giải thuật sắp xếp trộn chia toàn bộ mảng thành hai nửa. Tiến trình chia này tiếp tục diễn ra cho đến khi không còn chia được nữa và chúng ta thu được các giá trị tương ứng biểu diễn các phần tử trong mảng. Trong hình dưới, đầu tiên chúng ta chia mảng kích cỡ 8 thành hai mảng kích cỡ 4.

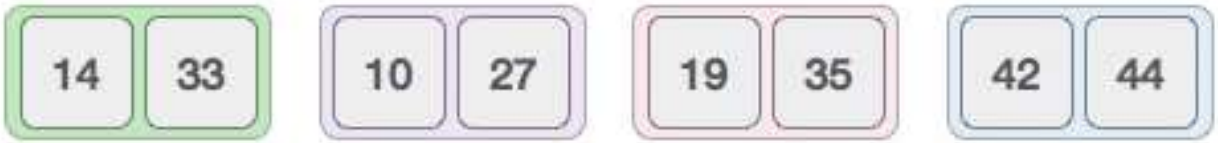
Tiến trình chia này không làm thay đổi thứ tự các phần tử trong mảng ban đầu. Bây giờ chúng ta tiếp tục chia các mảng này thành 2 nửa.

Tiến hành chia tiếp cho tới khi không còn chia được nữa.

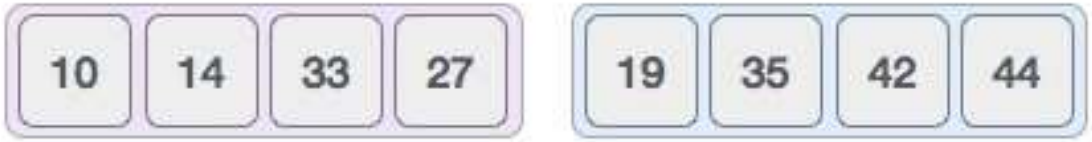


MEGRE SORT

Đầu tiên chúng ta so sánh hai phần tử trong mỗi list và sau đó tổ hợp chúng vào trong một list khác theo cách thức đã được sắp xếp. Ví dụ, 14 và 33 là trong các vị trí đã được sắp xếp. Chúng ta so sánh 27 và 10 và trong list khác chúng ta đặt 10 ở đầu và sau đó là 27. Tương tự, chúng ta thay đổi vị trí của 19 và 35. 42 và 44 được đặt tương ứng.



Vòng lặp tiếp theo là để kết hợp từng cặp list một ở trên. Chúng ta so sánh các giá trị và sau đó hợp nhất chúng lại vào trong một list chứa 4 giá trị, và 4 giá trị này đều đã được sắp thứ tự.



Sau bước kết hợp cuối cùng, danh sách sẽ trông giống như sau:

MEGRE SORT

- Ví dụ:

```
void MergeSort(int *p, int left, int right)
{
    if (right > left)
    {
        int mid;
        mid = (left + right) / 2;
        MergeSort(p, left, mid);
        MergeSort(p, mid + 1, right);
        Merge(p, left, mid, right);
    }
}
```

MEGRE SORT

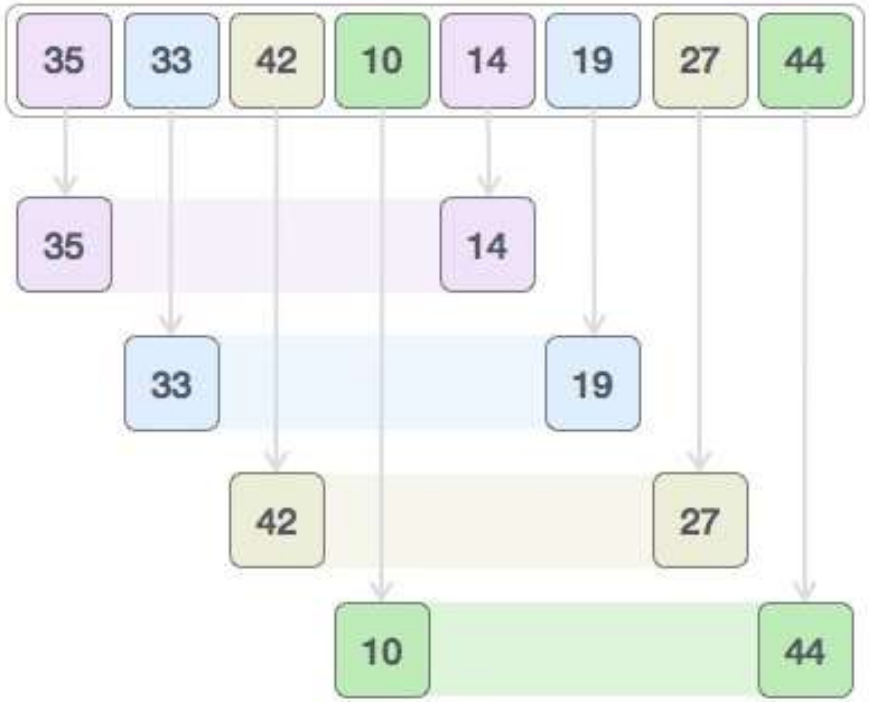
```
void Merge(int *p, int left, int mid, int right)
{
    int *temp, i = left, j = mid + 1;
    temp = new int[right - left + 1];
    for (int k = 0; k <= right - left; k++)
    {
        if (*(p + i) < *(p + j))
        {
            *(temp + k) = *(p + i);
            i++;
        }
        else
        {
            *(temp + k) = *(p + j);
            j++;
        }
        if (i == mid + 1)
        {
            while (j <= right)
            {
                k++;
                *(temp + k) = *(p + j);
                j++;
            }
            break;
        }
        if (j == right + 1)
        {
            while (i <= mid)
            {
                k++;
                *(temp + k) = *(p + i);
                i++;
            }
            break;
        }
    }
    for (int k = 0; k <= right - left; k++)
        *(p + left + k) = *(temp + k);
    delete temp;
}
```

SHELL SORT

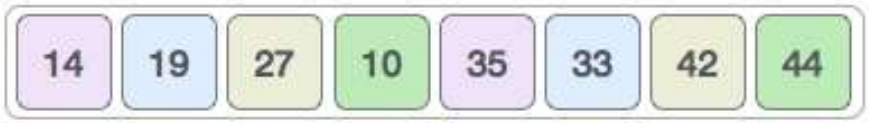
- Ý tưởng:
  - Sử dụng Selection Sort trên các phần tử có khoảng cách xa nhau, sau đó sắp xếp các phần tử có khoảng cách hẹp hơn;
  - Khoảng cách này là interval: là số vị trí từ phần tử này đến phần tử khác.
  - h = h * 3 + 1 (h là interval với giá trị ban đầu là 1).

SHELL SORT

- h = 4

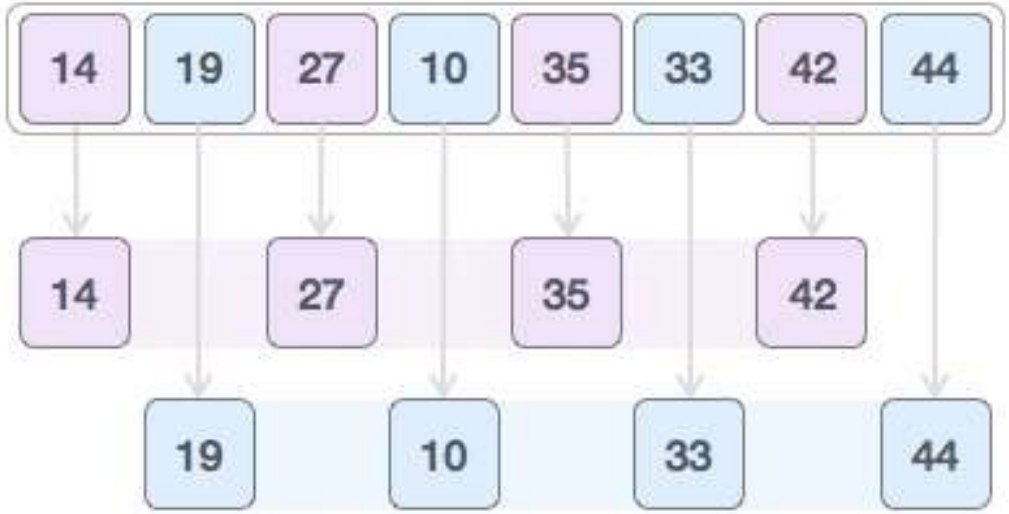


So sánh các giá trị này với nhau trong các danh sách con và tráo đổi chúng (nếu cần) trong mảng ban đầu. Sau bước này, mảng mới sẽ trống như sau:

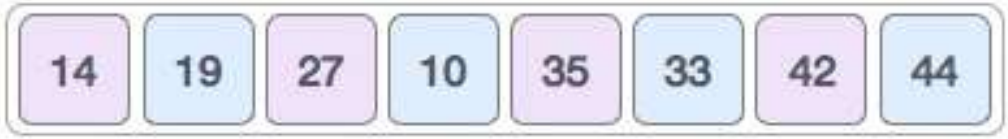


SHELL SORT

- h = 2



Tiếp tục so sánh và tráo đổi các giá trị (nếu cần) trong mảng ban đầu. Sau bước này, mảng sẽ trông như sau:



SHELL SORT

- h = 1

SHELL SORT

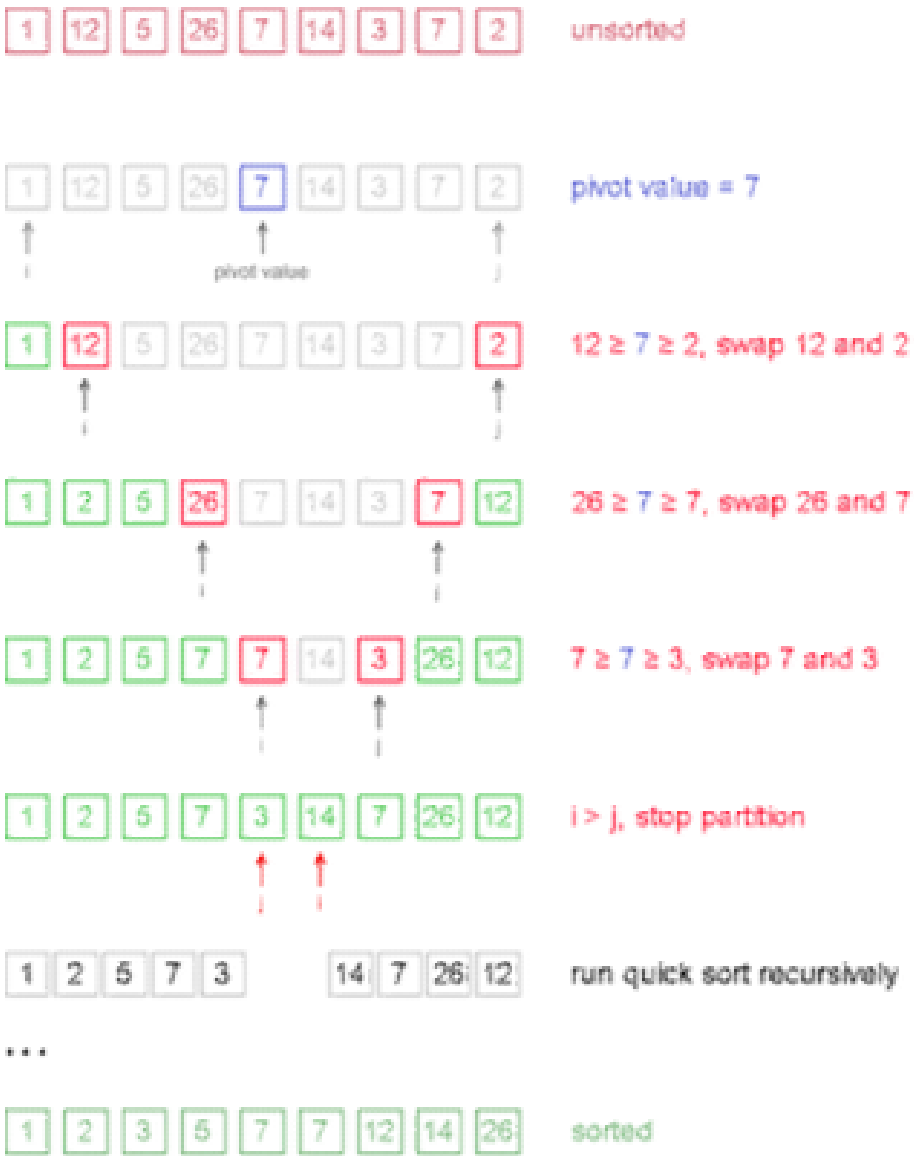
- Ví dụ:

```
void ShellSort(int *p, int length)
{
    for (int gap = length / 2; gap > 0; gap /= 2)
    {
        for (int i = gap; i < length; i += 1)
        {
            int temp = *(p + i);
            int j;
            for (j = i; j >= gap && *(p + j - gap) > temp; j -= gap)
                *(p + j) = *(p + j - gap);
            *(p + j) = temp;
        }
    }
}
```

QUICK SORT

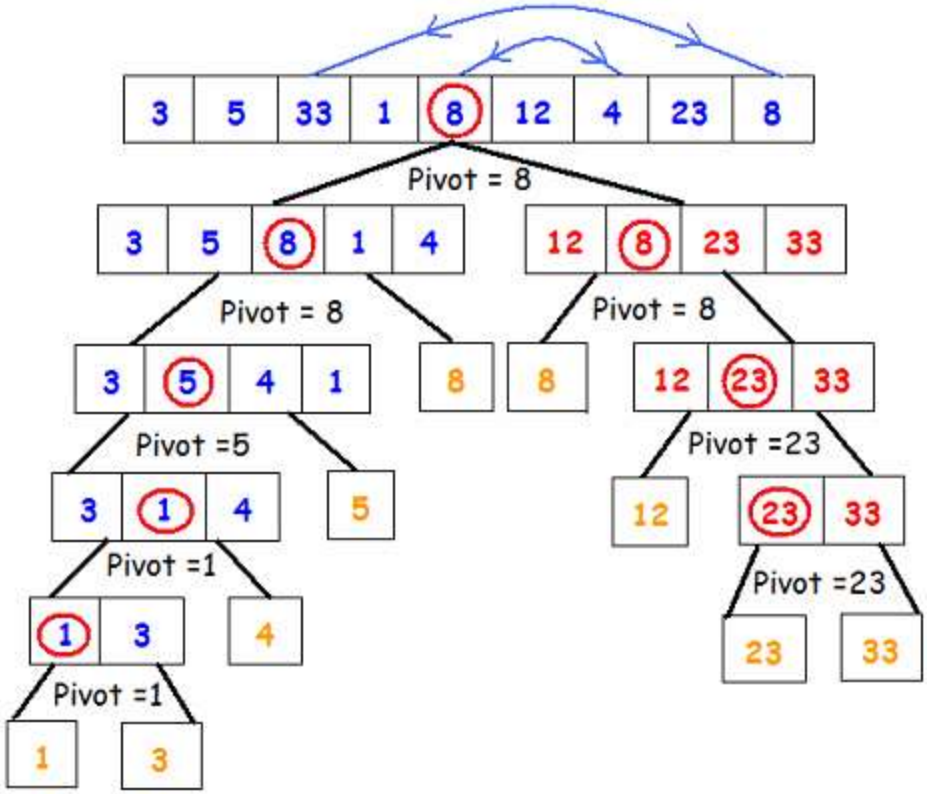
- Sắp xếp nhanh;
- Ý tưởng:
  - QuickSort chia mảng thành hai danh sách bằng cách so sánh từng phần tử của danh sách với một phần tử được chọn được gọi là phần tử chốt. Những phần tử nhỏ hơn hoặc bằng phần tử chốt được đưa về phía trước và nằm trong danh sách con thứ nhất, các phần tử lớn hơn chốt được đưa về phía sau và thuộc danh sách con thứ hai. Cứ tiếp tục chia như vậy tới khi các danh sách con đều có độ dài bằng 1.
- Cách chọn phần tử chốt:
  - Chọn phần tử đứng đầu hoặc đứng cuối;
  - Chọn phần tử đứng giữa mảng;
  - Chọn phần tử trung vị trong 3 phần tử đứng đầu, đứng giữa và đứng cuối;
  - Chọn phần tử ngẫu nhiên.

QUICK SORT


1 12 5 26 7 14 3 7 2
unsorted
1 12 5 26 7 14 3 7 2
pivot value = 7
i
pivot value
j
1 12 5 26 7 14 3 7 2
12 ≥ 7 ≥ 2, swap 12 and 2
i
j
1 2 5 26 7 14 3 7 12
26 ≥ 7 ≥ 7, swap 26 and 7
i
j
1 2 5 7 7 14 3 26 12
7 ≥ 7 ≥ 3, swap 7 and 3
i
j
1 2 5 7 3 14 7 26 12
i > j, stop partition
j
i
1 2 5 7 3
14 7 26 12
run quick sort recursively
...
1 2 3 5 7 7 12 14 26
sorted

QUICK SORT

- Ví dụ:



QUICK SORT

- Ví dụ:

```
void QuickSort(int *p, int left, int right)
{
    srand(time(NULL));
    int key = *(p + (left + rand() % (right - left + 1)));
    int i = left, j = right;
    while (i <= j)
    {
        while (*(p + i) < key) i++;
        while (*(p + j) > key) j--;
        if (i <= j)
        {
            if (i < j)
                Swap(*(p + i), *(p + j));
            i++; j--;
        }
    }
    if (left < j)
        QuickSort(p, left, j);
    if (i < right)
        QuickSort(p, i, right);
}
```

HEAP SORT

- Sắp xếp vun đống;
- Ý tưởng:
  - Phiên bản cải tiến của Selection Sort khi chia các phần tử thành 2 mảng con, 1 mảng các phần tử đã được sắp xếp và mảng còn lại các phần tử chưa được sắp xếp. Trong mảng chưa được sắp xếp, các phần tử lớn nhất sẽ được tách ra và đưa vào mảng đã được sắp xếp. Điều cải tiến ở Heapsort so với Selection sort ở việc sử dụng cấu trúc dữ liệu heap thay vì tìm kiếm tuyến tính (linear-time search) như Selection sort để tìm ra phần tử lớn nhất.

HEAP SORT

- Ví dụ:

```
void Heapify(int *p, int length, int i)
{
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < length && *(p + left) > *(p + largest))
        largest = left;
    if (right < length && *(p + right) > *(p + largest))
        largest = right;
    if (largest != i)
    {
        Swap(*(p + i), *(p + largest));
        Heapify(p, length, largest);
    }
}
```

HEAP SORT

- Ví dụ:

```
void HeapSort(int *p, int length)
{
    for (int i = length / 2 - 1; i >= 0; i--)
        Heapify(p, length, i);
    for (int i = length - 1; i >= 0; i--)
    {
        Swap(*p, *(p + i));
        Heapify(p, i, 0);
    }
}
```

Thank You !

IT FACULTY