Bài 1:

Thuật toán A: .

Thuật toán B: .

Xét hàm:

Với .

Do đó với thì hàm đồng biến.

Ta thấy .

Nên với thì .

thì hàm đồng biến.

thì .

Chọn thì .

Khi đó thuật toán A tốt hơn thuật toán B.

Bài 2:

#include <iostream>

using namespace std;

long long int pow(int a, int n);

int main(){

int n;

cin >> n;

cout << pow(2, n);

return 0;

}

long long int pow(int a, int n){

long long int result;

if(n == 0) return 1; // (1)

if(n % 2 == 0){ // (2)

result = pow(a, n/2); // (3)

return result\*result; // (4)

} else{

return pow(a, (n - 1)) \* a; // (5)

}

}

Tính độ phức tạp:

Giả sử T(n) là thời gian thực hiện hàm pow với độ lớn dữ liệu vào là n;

(1), (2), (4): O(1);

Với trường hợp n = 0 => Tốn O(1) => T(1) = C1;

Với n > 0;

Gọi thời gian thực hiện các phép toán cơ bản trong hàm là C2;

Với trường hợp n chẵn ta có // hàm pow ở dòng (3)

Với trường hợp n lẻ ta có // hàm pow ở dòng (5) với n – 1 là số chẵn

Do đó thời gian cần tính ở đây là (Do )

Cứ như vậy ta được công thức có dạng T(n) T() + 2.k.C2;

Khi Do 1 lẻ => Tính pow (a, 0) mất C1 thời gian;

Khi đó

* Độ phức tạp là

Bài 3:

Sắp xếp theo thứ tự tăng dần của // Từ trên xuống dưới sắp xếp theo thứ tự tăng dần

.

.

.

.

.

Bài 5:

Cho một danh sách gồm có n số, tìm cách xác định số lớn nhất và số nhỏ nhất của danh sách trong đó sử dụng ít hơn phép so sánh.

Xét trường hợp n chẵn:

Mảng a là mảng chứa n số;

Ta xét các cặp // k cặp

Có 2 biến min và max để lưu giá trị min và max của danh sách, được khởi tạo giá trị = a[0];

Nếu tồn tại cặp so sánh a[0] và a[1] tìm ra số lớn hơn và số bé hơn; -> mất 1 phép so sánh **(1)**

Còn lại k – 1 cặp:

Trong k – 1 cặp này ta cũng so sánh 2 số trong một cặp mất k – 1 phép so sánh. **(2)**

Xét trong mỗi cặp nằm trong k – 1 cặp này, ta tìm được số lớn hơn và số bé hơn.

Nếu số lớn hơn trong cặp này lớn hơn max hiện tại thì 1 phép so sánh

Nếu số nhỏ hơn trong cặp này nhỏ hơn min hiện tại thì 1 phép so sánh

Do ta có k – 1 cặp như thế nên số phép so sánh cần thực hiện nhằm gán lại biến max và min là 2.(k - 1) **(3)**

tổng số phép so sánh cần thực hiện là:

Trường hợp n lẻ:

Ta tìm max và min của dãy gồm số đầu tiên từ sau đó so sanh max và min tìm được với số cuối cùng;

Nếu n – 1 == 0 thì max = min = a[0]; -> Mất 1 phép so sánh: **(4)**

Trong trường hợp còn lại (n – 1 > 0)

Do n – 1 là số chẵn nên ta áp dụng phương pháp như trường hợp n là số chẵn cho

Khi đó số phép so sánh để tìm ra max và min của dãy n -1 số đầu tiên này là **(5)**

Sau đố so sánh giá trị max và min tìm được với giá trị còn lại cuối cùng của dãy để gán lại max và min nếu số còn lại > max hoặc < min. -> Mất 2 phép so sánh **(6)**

Tổng số phép so sánh trong trường hợp n lẻ này

Bài 6: Một mảng n\*n gồm các số 0 và 1, Tại mỗi dòng thì số 1 luôn đứng trước số 0. Tìm cách nhanh nhất để tìm dòng chứa nhiều số 1 nhất;

Dữ liệu đầu vào có dạng:

1 1 0 0 0

1 1 1 0 0

1 0 0 0 0

0 0 0 0 0

1 1 1 1 1

Cách tìm:

Gọi maxOne là số lượng số 1 của dòng có nhiều số 1 nhất trong dãy số;

Gọi maxRow là dòng chứa nhiều số 1 nhất

Ở dòng thứ nhất (chỉ số là 0) ta tìm vị trí đầu tiên mà số 0 xuất hiện // chi số trong danh sách tính bắt đầu từ 0

maxOne := vị trí này

maxRow := 1

Lặp lại điều sau cho các dòng còn lại (từ dòng chỉ số 1 -> n -1)

1. Kiểm tra giá trị tại vị trí maxOne ở dòng hiện tại, giá trị này là 0 hoặc 1:

2. Nếu giá trị này là 0, tức là dòng này có số lương số 1 <= dòng trước đó (do số 1 luôn đứng trước số 0),do đó ta chuyển sang dòng kế tiếp và lặp lại bước 1

3. Nếu giá trị này là 1, tức là số lượng số 1 ở dòng hiện tại > số lượng số 1 ở dòng phía trên, do đó gán giá trị maxRow := chỉ số dòng hiện tại + 1, ta tăng giá trị của maxOne lên 1,rồi kiểm tra giá trị tại vị trí maxOne mới(sau khi tăng 1), nếu vẫn là số 1 thì lại tăng maxOne lên 1, cứ tăng maxOne lên 1 và kiểm tra như vậy cứ cho đến khi gặp giá trị tại vị trí maxOne là 0 hoặc khi maxOne = n.

- Nếu maxOne = n tức là dòng này có n số 1, là giá trị số lượng số 1 lớn nhất có thể đạt được được nên dừng tìm kiếm, và trả về giá trị maxRow

- Nếu ta tìm được giá trị tại vị trí maxOne mới ở dòng hiện tại là 0 thì ta chuyển sang dòng kế tiếp và lặp lại bước 1.

4. Trả về giá trị maxRow

Chương trình minh họa:

#include <iostream>

using namespace std;

int findMaxRow(int a[5][5], int n);

int main(){

int a[5][5] = {{0, 0, 0, 0, 0},

{1, 1, 1, 1, 1},

{1, 1, 1, 1, 1},

{1, 0, 0, 0, 0},

{0, 0, 0, 0, 0}};

cout << findMaxRow(a, 5);

}

int findMaxRow(int a[5][5], int n){

int maxOne = 0;

int maxRow = 1;

for(int i = 0; i < n; i++){

if(a[0][i] == 1) maxOne++;

}

for(int i = 1; i < n; i++){ // (1)

if(a[i][maxOne] == 0) continue;

if(a[i][maxOne] == 1){

maxRow = i + 1;

maxOne++;

while(a[i][maxOne] == 1 && maxOne < n){ // (2)

maxOne++;

}

if(maxOne == n) return maxRow;

}

}

return maxRow;

}

Tính độ phức tạp:

Trong trường hợp xấu nhất chúng ta phải

kiểm tra tất tất cả n dòng

và giá trị maxOne phải tăng lên từ 0 -> n;

Tức là chỉ có duy nhất 1 dòng chứa toàn các số 1 ở vị trí cuối cùng (dòng thứ n)

Do ta lặp lại việc kiểm tra giá trị phần tử của mỗi dòng tại vị trí maxOne n lần (do có n dòng) nên độ phức tạp là O(n); (trong vòng for vị trí (1) nhưng chưa tính vòng while)

maxOne mỗi lần chỉ tăng 1, mà nó tăng từ 0-> n (do có n cột) để thực hiện điều này -> dùng vòng lặp mất O(n) qua tất cả các lần (// được thực hiện thông qua vòng while vị trí 2)

Vậy độ phức tạp là O(n + n) = O(n);

Có thể thấy rõ hơn trong trường hợp, tất cả các dòng phía trên = 0, dòng cuối cùng tất cả = 1;

Từ 0 -> n – 2 thì trong vòng for vị trí số (1) có độ phức tạp là O(1) -> thực hiên n – 1 lần -> mất O(n)

Khi ở vòng for : với i = n – 1 thì tại dòng cuối toàn số 1, bên trong phần lệnh while có độ phức tạp O(1) được thực hiên n lần -> O(n)

Do đó độ phức tạp là O(n + n) = O(n);