



TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG



Cấu trúc dữ liệu và thuật toán

Nguyễn Khánh Phương

Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn

Nội dung khóa học

Chương 1. Các khái niệm cơ bản

Chương 2. Các sơ đồ thuật toán

Chương 3. Các cấu trúc dữ liệu cơ bản

Chương 4. Cây

Chương 5. Sắp xếp

Chương 6. Tìm kiếm

Chương 7. Đồ thị



TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG



Chương 1. Các khái niệm cơ bản

Nguyễn Khánh Phương

**Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn**

Nội dung

- 1.1. Ví dụ mở đầu
- 1.2. Thuật toán và độ phức tạp
- 1.3. Kí hiệu tiệm cận
- 1.4. Giả ngôn ngữ (Pseudo code)
- 1.5. Một số kĩ thuật phân tích thuật toán
- 1.6. Giải công thức đệ quy

Nội dung

1.1. Ví dụ mở đầu

1.2. Thuật toán và độ phức tạp

1.3. Kí hiệu tiệm cận

1.4. Giả ngôn ngữ (Pseudo code)

1.5. Một số kĩ thuật phân tích thuật toán

1.6. Giải công thức đệ quy

Ví dụ: Tìm dãy con lớn nhất (The maximum subarray problem)

- Cho dãy số gồm n số:

$$a_1, a_2, \dots, a_n$$

Dãy gồm liên tiếp các số a_i, a_{i+1}, \dots, a_j với $1 \leq i \leq j \leq n$ được gọi là **dãy con** của dãy đã cho và $\sum_{k=i}^j a_k$ được gọi là *trọng lượng của dãy con này*

Bài toán đặt ra là: Hãy tìm trọng lượng lớn nhất của các dãy con, tức là tìm cực đại giá trị $\sum_{k=i}^j a_k$. Ta gọi dãy con có trọng lượng lớn nhất là **dãy con lớn nhất.**

Ví dụ: Cho dãy số -2, 11, -4, 13, -5, 2 thì cần đưa ra câu trả lời là 20 (dãy con lớn nhất là 11, -4, 13 với giá trị $= 11 + (-4) + 13 = 20$)

➔ Để giải bài toán này, ta có thể dùng nhiều cách khác nhau, ví dụ duyệt toán bộ (brute force), chia để trị (divide and conquer), quy hoạch động (dynamic programming), ...

Ví dụ: Tìm dãy con lớn nhất (The maximum subarray problem)

1.1.1. Duyệt toàn bộ (Brute force)

1.1.2. Duyệt toàn bộ có cải tiến

1.1.3 Thuật toán đệ quy (Recursive algorithm)

1.1.4. Thuật toán quy hoạch động (Dynamic programming)

Ví dụ: Tìm dãy con lớn nhất (The maximum subarray problem)

1.1.1. Duyệt toàn bộ (Brute force)

1.1.2. Duyệt toàn bộ có cải tiến

1.1.3 Thuật toán đệ quy (Recursive algorithm)

1.1.4. Thuật toán quy hoạch động (Dynamic programming)

1.1.1. Thuật toán duyệt toàn bộ giải bài toán dãy con lớn nhất

- Thuật toán đơn giản đầu tiên có thể nghĩ để giải bài toán đặt ra là: Duyệt tất cả các dãy con có thể :

$$a_i, a_{i+1}, \dots, a_j \quad \text{với} \quad 1 \leq i \leq j \leq n,$$

và tính tổng của mỗi dãy con để tìm ra trọng lượng lớn nhất.

- Trước hết nhận thấy rằng, tổng số các dãy con có thể của dãy đã cho là:

$$C(n, 1) + C(n, 2) = n^2/2 + n/2$$

Ví dụ ứng dụng

- Giả sử ta biết giá của cổ phiếu của công ty A từ ngày i đến ngày j như sau:

Ngày	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Giá	100	112	109	85	105	102	86	63	81	101	94	106	101	79	93	89	95

- Ta cần mua một số cổ phiếu, **duy nhất 1 lần**, rồi bán ra tại một ngày nào đó sau đây.
- Làm thế nào để tối đa hóa lợi nhuận ?
 - Chiến lược: mua vào lúc giá thấp, bán ra lúc giá cao [buy low, sell high] không phải lúc nào cũng cho lợi nhuận cao nhất

Ví dụ: Cho giá cổ phiếu như ở đồ thị. Ta thu được lợi nhuận cao nhất là 3\$ nếu mua vào ở thứ 2 với giá 7\$, và bán ra ở ngày thứ 3 với giá 10\$. Giá 7\$ mua vào ở ngày 2 không phải là giá thấp nhất, và giá 10\$ bán ra ở ngày 3 cũng không phải là giá cao nhất



- Lời giải: Ta dễ dàng tìm được bằng cách duyệt hết tất cả các khả năng:
 - Có bao nhiêu cặp ngày mua/bán có thể có trong khoảng thời gian n ngày?
 - Tính lợi nhuận thu được cho mỗi cặp ngày, để tìm cặp ngày có lợi nhuận cao nhất
- Liệu ta có thể làm tốt hơn hay không?
 - Câu trả lời: Có, bằng cách quy về bài toán dãy con lớn nhất

Quy về bài toán dãy con lớn nhất

Ngày	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Giá	100	112	109	85	105	102	86	63	81	101	94	106	101	79	93	89	95

- Tìm dãy các ngày liên tiếp sao cho:
 - Tổng lượng giá thay đổi ở ngày cuối so với ngày đầu là lớn nhất
- Nhìn vào giá của từng ngày
 - Lượng giá thay thay đổi vào ngày i : giá cổ phiếu ngày i trừ đi giá cổ phiếu ngày $i-1$
 - Ta có mảng giá thay đổi như sau:

Ngày	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Giá	100	112	109	85	105	102	86	63	81	101	94	106	101	79	93	89	95
Thay đổi		12	-3	-24	20	-3	-16	-23	18	20	-7	12	-5	-22	14	-4	6

- Tìm dãy con liên tiếp có tổng lớn nhất

- **Dãy con lớn nhất**

e.g.: 12,-3,-24,20,-3,-16,-23,**18,20,-7,12**,-5,-22,14,-4,6

Dãy con lớn nhất: $18+20+(-7)+12 = 43$

- ➔ Mua sau ngày thứ 7, bán ra sau ngày thứ 11: mua với giá = 63, bán ra với lúc = 106
- ➔ Lợi nhuận = $106-63=43$

Thuật toán duyệt toàn bộ: duyệt tất cả các dãy con

Chỉ số i	0	1	2	3	4	5
a[i]	-2	11	-4	13	-5	2

i = 0: (-2), (-2, 11), (-2, 11, -4), (-2, 11, -4, 13), (-2, 11, -4, 13, -5), (-2, 11, -4, 13, -5, 2)

i = 1: (11), (11, -4), (11, -4, 13), (11, -4, 13, -5), (11, -4, 13, -5, 2)

i = 2: (-4), (-4, 13), (-4, 13, -5), (-4, 13, -5, 2)

i = 3: (13), (13, -5), (13, -5, 2)

i = 4: (-5), (-5, 2)

i = 5: (2)

Với mỗi chỉ số i : duyệt tất cả các dãy con bắt đầu từ phần tử a[i] có 1 phần tử, có 2 phần tử, ... :

- i = 0: tất cả các dãy con bắt đầu từ a[0] có 1 phần tử, 2 phần tử, ... 6 phần tử
- i = 1: tất cả các dãy con bắt đầu từ a[1] có 1 phần tử, 2 phần tử, ... 5 phần tử
- ...
- i = 5: tất cả các dãy con bắt đầu từ a[5] có 1 phần tử

```
int maxSum = 0;
for (int i=0; i<n; i++) {
    for (int j=i; j<n; j++) {
        int sum = 0;
        for (int k=i; k<=j; k++)
            sum += a[k];
        if (sum > maxSum)
            maxSum = sum;
    }
}
```

Thuật toán duyệt toàn bộ: duyệt tất cả các dãy con

- Phân tích thuật toán:** Ta sẽ tính số lượng phép cộng mà thuật toán phải thực hiện, tức là đếm xem dòng lệnh

sum += a[k]

phải thực hiện bao nhiêu lần. Số lượng phép cộng là:

$$\begin{aligned}\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j-i+1) &= \sum_{i=0}^{n-1} (1+2+\dots+(n-i)) = \sum_{i=0}^{n-1} \frac{(n-i)(n-i+1)}{2} \\ &= \frac{1}{2} \sum_{k=1}^n k(k+1) = \frac{1}{2} \left[\sum_{k=1}^n k^2 + \sum_{k=1}^n k \right] = \frac{1}{2} \left[\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2} \right] \\ &= \frac{n^3}{6} + \frac{n^2}{2} + \frac{n}{3}\end{aligned}$$

```
int maxSum = 0;
for (int i=0; i<n; i++) {
    for (int j=i; j<n; j++) {
        int sum = 0;
        for (int k=i; k<=j; k++)
            sum += a[k];
        if (sum > maxSum)
            maxSum = sum;
    }
}
```

Ví dụ: Tìm dãy con lớn nhất (The maximum subarray problem)

1.1.1. Duyệt toàn bộ (Brute force)

1.1.2. Duyệt toàn bộ có cải tiến

1.1.3 Thuật toán đệ quy (Recursive algorithm)

1.1.4. Thuật toán quy hoạch động (Dynamic programming)

1.1.2. Duyệt toàn bộ có cải tiến

Thuật toán duyệt toàn bộ: duyệt tất cả các dãy con

Chỉ số i	0	1	2	3	4	5
a[i]	-2	11	-4	13	-5	2

i = 0: (-2), (-2, 11), (-2, 11, -4), (-2, 11, -4, 13), (-2, 11, -4, 13, -5), (-2, 11, -4, 13, -5, 2)

i = 1: (11), (11, -4), (11, -4, 13), (11, -4, 13, -5), (11, -4, 13, -5, 2)

i = 2: (-4), (-4, 13), (-4, 13, -5), (-4, 13, -5, 2)

i = 3: (13), (13, -5), (13, -5, 2)

i = 4: (-5), (-5, 2)

i = 5: (2)

Nhận thấy, ta có thể tính tổng các phần tử từ vị trí i đến j từ tổng của các phần tử từ i đến $j-1$ chỉ bằng 1 phép cộng:

$$\sum_{k=i}^j a[k] = a[j] + \sum_{k=i}^{j-1} a[k]$$

Tổng các phần tử từ i đến j

Tổng các phần tử từ i đến $j-1$

```
int maxSum = 0;
for (int i=0; i<n; i++) {
    for (int j=i; j<n; j++) {
        int sum = 0;
        for (int k=i; k<=j; k++)
            sum += a[k];
        if (sum > maxSum)
            maxSum = sum;
    }
}
```



```
int maxSum = a[0];
for (int i=0; i<n; i++) {
    int sum = 0;
    for (int j=i; j<n; j++) {
        sum += a[j];
        if (sum > maxSum)
            maxSum = sum;
    }
}
```

1.1.2. Duyệt toàn bộ có cải tiến

Thuật toán duyệt toàn bộ: duyệt tất cả các dãy con

- Cài đặt cải tiến:**

Nhận thấy, ta có thể tính tổng các phần tử từ vị trí i đến j từ tổng của các phần tử từ i đến $j-1$ chỉ bằng 1 phép cộng:

$$\underbrace{\sum_{k=i}^j a[k]}_{\text{Tổng các phần tử từ } i \text{ đến } j} = a[j] + \underbrace{\sum_{k=i}^{j-1} a[k]}_{\text{Tổng các phần tử từ } i \text{ đến } j-1}$$

Tổng các phần tử từ i đến j

Tổng các phần tử từ i đến $j-1$

```
int maxSum = 0;
for (int i=0; i<n; i++) {
    for (int j=i; j<n; j++) {
        int sum = 0;
        for (int k=i; k<=j; k++)
            sum += a[k];
        if (sum > maxSum)
            maxSum = sum;
    }
}
```



```
int maxSum = a[0];
for (int i=0; i<n; i++) {
    int sum = 0;
    for (int j=i; j<n; j++) {
        sum += a[j];
        if (sum > maxSum)
            maxSum = sum;
    }
}
```


1.1.2. Duyệt toàn bộ có cải tiến

Thuật toán duyệt toàn bộ: duyệt tất cả các dãy con

- **Phân tích thuật toán.** Ta lại tính số lần thực hiện phép cộng:

`Sum += a[j]`

Và thu được kết quả:

$$\sum_{i=0}^{n-1} (n-i) = n + (n-1) + \dots + 1 = \frac{n^2}{2} + \frac{n}{2}$$

- Để ý rằng số này là đúng bằng số lượng dãy con. Dường như thuật toán thu được là rất tốt, vì ta phải xét mỗi dãy con đúng 1 lần.

```
int maxSum = a[0];
for (int i=0; i<n; i++) {
    int sum = 0;
    for (int j=i; j<n; j++) {
        sum += a[j];
        if (sum > maxSum)
            maxSum = sum;
    }
}
```

Ví dụ: Tìm dãy con lớn nhất (The maximum subarray problem)

1.1.1. Duyệt toàn bộ (Brute force)

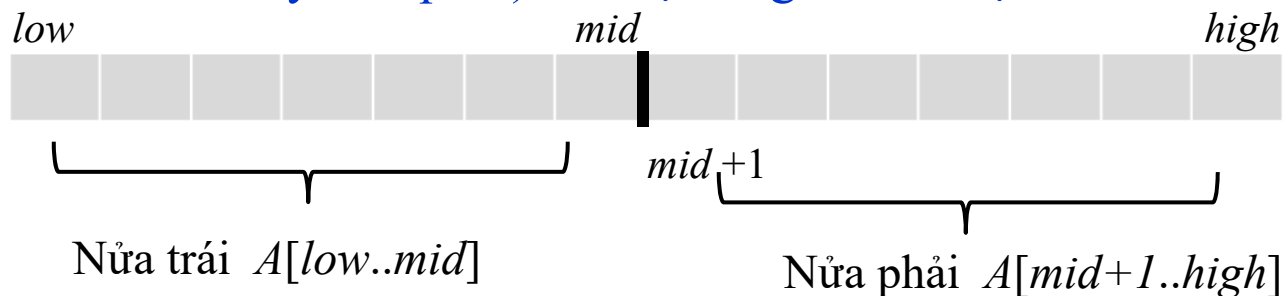
1.1.2. Duyệt toàn bộ có cải tiến

1.1.3 Thuật toán đệ quy (Recursive algorithm)

1.1.4. Thuật toán quy hoạch động (Dynamic programming)

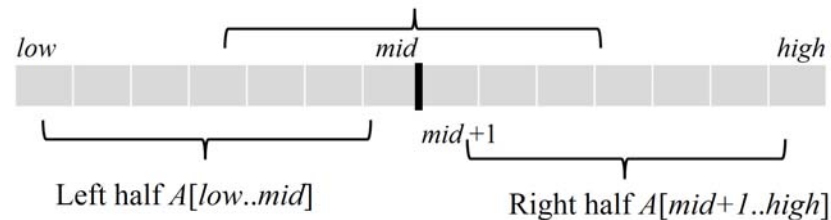
1.1.3. Thuật toán đệ quy giải bài toán dãy con lớn nhất

- Ta còn có thể xây dựng thuật toán tốt hơn nữa bằng kỹ thuật chia để trị (divide-and-conquer). Kỹ thuật này bao gồm các bước sau:
 - Chia bài toán cần giải ra thành các bài toán con cùng dạng
 - Giải mỗi bài toán con một cách đệ quy
 - *Trường hợp cơ sở: khi bài toán con có kích thước đủ nhỏ, ta giải nó bằng phương pháp duyệt toàn bộ.*
 - Tổ hợp lời giải của các bài toán con để thu được lời giải của bài toán xuất phát.
- Để giải bài toán dãy con lớn nhất:
 - Sử dụng phân tử chính giữa để chia đôi dãy đã cho thành hai dãy con (gọi tắt là dãy nửa trái và dãy nửa phải) với độ dài giảm đi một nửa



1.1.3. Thuật toán đệ quy giải bài toán dãy con lớn nhất

- Để tổ hợp lời giải, nhận thấy rằng dãy con lớn nhất $A[i..j]$ của dãy $A[low..high]$ chỉ có thể xảy ra một trong 3 trường hợp (với $mid = \lfloor (low+high)/2 \rfloor$):
 - Dãy con lớn nhất nằm ở dãy nửa bên trái $A[low..mid]$ ($low \leq i \leq j \leq mid$)
 - Dãy con lớn nhất nằm ở dãy nửa bên phải $A[mid+1..high]$ ($mid < i \leq j \leq high$)
 - Giao điểm giữa mid ($low \leq i \leq mid < j \leq high$) {dãy con lớn nhất bắt đầu ở nửa trái và kết thúc ở nửa phải}

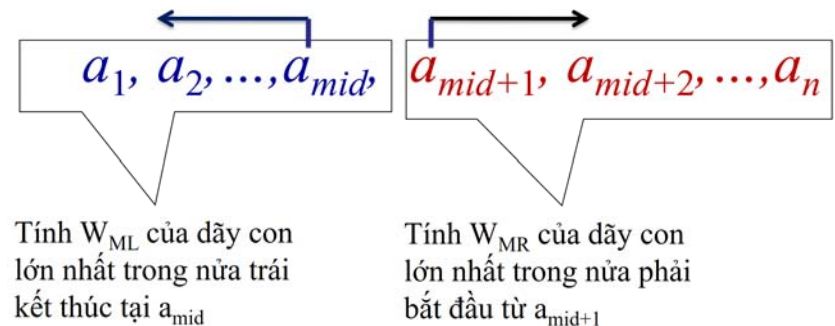


- Do đó, nếu kí hiệu trọng lượng của dãy con lớn nhất ở nửa trái là w_L , ở nửa phải là w_R , và giao điểm giữa là w_M , thì trọng lượng lớn nhất cần tìm là $\max(w_L, w_R, w_M)$

```
MaxSub(a, low, high);
{
    if (low = high) return a[low] //base case: only 1 element
    else
    {
        mid = (low+high)/2;
        wL = MaxSub(a, low, mid);
        wR = MaxSub(a, mid+1, high);
        wM = MaxCrossMidPoint(a, low, mid, high);
        return max(wL, wR, wM);
    }
}
```

1.1.3. Thuật toán đệ quy giải bài toán dãy con lớn nhất

- Việc tìm trọng lượng của dãy con lớn nhất ở nửa trái (w_L) và nửa phải (w_R) có thể thực hiện một cách đệ quy:
 - Trường hợp cơ sở (base case): dãy nửa trái / phải chỉ gồm duy nhất 1 phần tử
- Để tìm trọng lượng w_M của dãy con lớn nhất bắt đầu từ nửa trái và kết thúc ở nửa phải, ta thực hiện như sau:
 - Ở dãy nửa trái: tìm trọng lượng w_{ML} của dãy con lớn nhất kết thúc ở điểm chia mid
 - Ở dãy nửa phải: tính trọng lượng w_{MR} của dãy con lớn nhất bắt đầu ở vị trí mid+1
 - Khi đó $w_M = w_{ML} + w_{MR}$.



```
MaxSub(a, low, high);
{
    if (low == high) return a[low] //base case: only 1 element
    else
    {
        mid = (low+high)/2;
        wL = MaxSub(a, low, mid);
        wR = MaxSub(a, mid+1, high);
        wM = MaxCrossMidPoint(a, low, mid, high);
        return max(wL, wR, wM);
    }
}
```

Ví dụ: tính trọng lượng w_M của dãy con lớn nhất bắt đầu ở dãy nửa trái và kết thúc ở dãy nửa phải

mid = 5

Ở dãy nửa trái:

	1	2	3	4	5	6	7	8	9	10
A	13	-3	-25	20	-3	-16	-23	18	20	-7

$$\begin{aligned}
 S[5..5] &= -3 \\
 S[4..5] &= 17 \leftarrow (\text{max_left} = 4) \\
 S[3..5] &= -8 \\
 S[2..5] &= -11 \\
 S[1..5] &= 2
 \end{aligned}$$

mid = 5

Ở dãy nửa phải:

	1	2	3	4	5	6	7	8	9	10
A	13	-3	-25	20	-3	-16	-23	18	20	-7

$$\begin{aligned}
 S[6..6] &= -16 \\
 S[6..7] &= -39 \\
 S[6..8] &= -21 \\
 S[6..9] &= (\text{max_right} = 9) \Rightarrow -1 \\
 S[6..10] &= -8
 \end{aligned}$$

\Rightarrow Dãy con lớn nhất giao điểm chia *mid* là $S[4..9] = 17 + (-1) = 16$

Tính trọng lượng w_M của dãy con lớn nhất bắt đầu ở dãy nửa trái và kết thúc ở dãy nửa phải


- Ở dãy nửa trái: tìm trọng lượng w_{ML} của dãy con lớn nhất kết thúc ở điểm chia mid
- Ở dãy nửa phải: tính trọng lượng w_{MR} của dãy con lớn nhất bắt đầu ở vị trí mid+1

```
MaxLeft(a, low, mid);  
{  
    maxSum = -∞; sum = 0;  
    for (int k=mid; k>=low; k--) {  
        sum = sum+a[k];  
        maxSum = max(sum, maxSum);  
    }  
    return maxSum;  
}
```

```
MaxRight(a, mid, high);  
{  
    maxSum = -∞; sum = 0;  
    for (int k=mid; k<=high; k++){  
        sum = sum+a[k];  
        maxSum = max(sum, maxSum);  
    }  
    return maxSum;  
}
```

- Khi đó $w_M = w_{ML} + w_{MR}$.

```
MaxSub(a, low, high);  
{  
    if (low = high) return a[low] //base case: only 1 element  
    else  
    {  
        mid = (low+high)/2;  
        wL = MaxSub(a, low, mid);  
        wR = MaxSub(a, mid+1, high);  
        wM = MaxCrossMidPoint(a, low, mid, high);  
        return max(wL, wR, wM);  
    }  
}
```

 $w_M = \text{MaxLeft}(a, \text{low}, \text{mid}) + \text{MaxRight}(a, \text{mid}, \text{high});$

1.1.3. Thuật toán đệ quy giải bài toán dãy con lớn nhất

Phân tích thuật toán: Ta cần tính xem lệnh gọi $\text{MaxSub}(a, 1, n)$ để thực hiện thuật toán đòi hỏi bao nhiêu phép cộng?

- Thủ tục MaxLeft và MaxRight yêu cầu

$$n/2 + n/2 = n \text{ phép cộng}$$

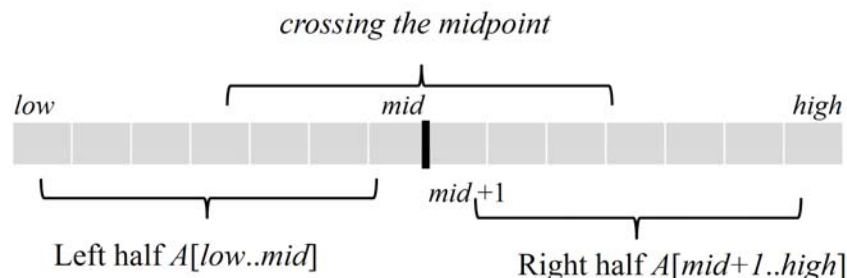
- Vì vậy, nếu gọi $T(n)$ là số phép cộng mà lệnh $\text{maxSub}(a, 1, n)$ cần thực hiện, ta có công thức đệ quy sau:

$$T(n) = \begin{cases} 0 & n = 1 \\ T(\frac{n}{2}) + T(\frac{n}{2}) + n = 2T(\frac{n}{2}) + n & n > 1 \end{cases}$$

Giải công thức đệ quy này, ta có $T(n) = n \log n$

```
MaxLeft(a, low, mid);
{
    maxSum = -∞; sum = 0;
    for (int k=mid; k>=low; k--) {
        sum = sum+a[k];
        maxSum = max(sum, maxSum);
    }
    return maxSum;
}
MaxRight(a, mid, high);
{
    maxSum = -∞; sum = 0;
    for (int k=mid; k<=high; k++){
        sum = sum+a[k];
        maxSum = max(sum, maxSum);
    }
    return maxSum;
}
```

```
MaxSub(a, low, high);
{
    if (low = high) return a[low] //base case: only 1 element
    else
    {
        mid = (low+high)/2;
        wL = MaxSub(a, low, mid);
        wR = MaxSub(a, mid+1, high);
        wM = MaxLeft(a, low, mid) + MaxRight(a, mid, high);
        return max(wL, wR, wM);
    }
}
```



1.1.3. Thuật toán đệ quy giải bài toán dãy con lớn nhất

Ví dụ: để tìm dãy con lớn nhất cho mảng a gồm 10 phần tử cho ở bảng dưới đây

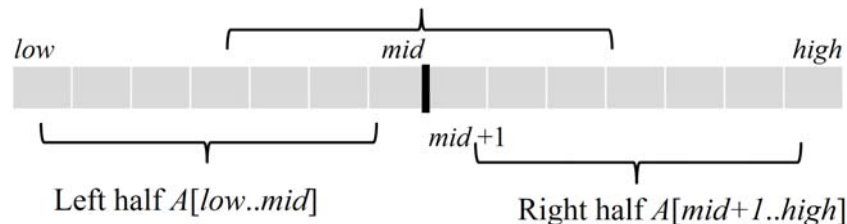
1	2	3	4	5	6	7	8	9	10
13	-3	-25	20	-3	-16	-23	18	20	-7

Ta sẽ phải gọi lệnh: `MaxSub(a, 1, 10)`

```
MaxLeft(a, low, mid);
{
    maxSum = -∞; sum = 0;
    for (int k=mid; k>=low; k--) {
        sum = sum+a[k];
        maxSum = max(sum, maxSum);
    }
    return maxSum;
}

MaxRight(a, mid, high);
{
    maxSum = -∞; sum = 0;
    for (int k=mid; k<=high; k++){
        sum = sum+a[k];
        maxSum = max(sum, maxSum);
    }
    return maxSum;
}
```

```
MaxSub(a, low, high);
{
    if (low = high) return a[low] //base case: only 1 element
    else
    {
        mid = (low+high)/2;
        wL = MaxSub(a, low, mid);
        wR = MaxSub(a, mid+1, high);
        wM = MaxLeft(a, low, mid) + MaxRight(a, mid, high);
        return max(wL, wR, wM);
    }
}
```



Bài toán dãy con lớn nhất: so sánh thời gian tính của các thuật toán

Số lượng phép cộng mà mỗi thuật toán cần thực hiện là:

1.1.1. Duyệt toàn bộ $\frac{n^3}{6} + \frac{n^2}{2} + \frac{n}{3}$

1.1.2. Duyệt toàn bộ có cải tiến $\frac{n^2}{2} + \frac{n}{2}$

1.1.3 Thuật toán đệ quy: $n \log n$

➔ Cùng một bài toán, ta đã đề xuất 3 thuật toán đòi hỏi số lượng phép toán khác nhau, và vì thế sẽ đòi hỏi thời gian tính khác nhau.

Bảng dưới đây cho thấy thời gian tính của 3 thuật toán trên, với giả thiết: máy tính có thể thực hiện 10^8 phép cộng trong một giây

Độ phức tạp	n=10	Thời gian	n=100	Thời gian	n=10 ⁴	Thời gian	n=10 ⁶	Thời gian
n^3	10 ³	10 ⁻⁵ giây	10 ⁶	10 ⁻² giây	10 ¹²	2.7 hours	10 ¹⁸	115 ngày
n^2	100	10 ⁻⁶ giây	10000	10 ⁻⁴ giây	10 ⁸	1 giây	10 ¹²	2.7 giờ
$n \log n$	33.2	3.3*10 ⁻⁸ giây	664	6.6*10 ⁻⁶ giây	1.33*10 ⁵	10 ⁻³ giây	1.99*10 ⁷	2*10 ⁻¹ sec
e^n	2.2*10 ⁴	1*10 ⁻⁴ giây	2.69*10 ⁴³	>10 ²⁶ thế kỉ	8.81*10 ⁴³⁴²	>10 ⁴³²⁷ thế kỉ		

Bài toán dãy con lớn nhất: so sánh thời gian tính của các thuật toán

Độ phức tạp	n=10	Thời gian	n=100	Thời gian	n=10 ⁴	Thời gian	n=10 ⁶	Thời gian
n^3	10 ³	10 ⁻⁵ giây	10 ⁶	10 ⁻² giây	10 ¹²	2.7 hours	10 ¹⁸	115 ngày
n^2	100	10 ⁻⁶ giây	10000	10 ⁻⁴ giây	10 ⁸	1 giây	10 ¹²	2.7 giờ
$n \log n$	33.2	3.3*10 ⁻⁸ giây	664	6.6*10 ⁻⁶ giây	1.33*10 ⁵	10 ⁻³ giây	1.99*10 ⁷	2*10 ⁻¹ sec
e^n	2.2*10 ⁴	1*10 ⁻⁴ giây	2.69*10 ⁴³	>10 ²⁶ thế kỉ	8.81*10 ⁴³⁴²	>10 ⁴³²⁷ thế kỉ		

- Với n nhỏ thời gian tính là không đáng kể.
- Vấn đề trở nên nghiêm trọng hơn khi $n > 10^6$. Lúc đó chỉ có thuật toán thứ ba (thời gian $n \log n$) là có thể áp dụng được trong thời gian thực.
- Còn có thể làm tốt hơn nữa không?
 - Yes! Có thể đề xuất thuật toán chỉ đòi hỏi n phép cộng!

Ví dụ: Tìm dãy con lớn nhất (The maximum subarray problem)

1.1.1. Duyệt toàn bộ (Brute force)

$$\frac{n^3}{6} + \frac{n^2}{2} + \frac{n}{3}$$

1.1.2. Duyệt toàn bộ có cải tiến

$$\frac{n^2}{2} + \frac{n}{2}$$

1.1.3. Thuật toán đệ quy (Recursive algorithm)

$$n \log n$$

1.1.4. Thuật toán Quy hoạch động (Dynamic programming)

$$n$$

1.1.4. Thuật toán quy hoạch động giải bài toán dãy con lớn nhất

Thuật toán quy hoạch động được chia làm 3 giai đoạn:

1. **Phân rã (Divide):** chia bài toán cần giải thành những bài toán con (Bài toán con (Subproblem): là bài toán có cùng dạng với bài toán đã cho, nhưng kích thước nhỏ hơn)
2. **Ghi nhận lời giải:** lưu trữ lời giải của các bài toán con vào 1 bảng
3. **Tổng hợp lời giải:** Lần lượt từ lời giải của các bài toán con kích thước nhỏ hơn tìm cách xây dựng lời giải của bài toán kích thước lớn hơn, cho đến khi thu được lời giải của **bài toán xuất phát (là bài toán con có kích thước lớn nhất)**.

1.1.4. Thuật toán quy hoạch động giải bài toán dãy con lớn nhất

Thuật toán quy hoạch động được chia làm 3 giai đoạn:

1. Phân rã:

- Gọi s_i là trọng lượng của dãy con lớn nhất của dãy $a_1, a_2, \dots, a_i, i = 1, 2, \dots, n$.
- Rõ ràng, s_n là giá trị cần tìm (lời giải của bài toán).

3. Tổng hợp lời giải:

- $s_1 = a_1$
- Giả sử $i > 1$ và ta đã biết giá trị s_k với $k = 1, 2, \dots, i-1$. Ta cần tính giá trị s_i là trọng lượng của dãy con lớn nhất của dãy:
$$a_1, a_2, \dots, a_{i-1}, a_i.$$
- Nhận thấy rằng: dãy con lớn nhất của dãy $a_1, a_2, \dots, a_{i-1}, a_i$ có thể hoặc bao gồm phần tử a_i hoặc không bao gồm phần tử a_i → do đó, dãy con lớn nhất của dãy $a_1, a_2, \dots, a_{i-1}, a_i$ chỉ có thể là một trong 2 dãy sau:
 - Dãy con lớn nhất của dãy a_1, a_2, \dots, a_{i-1}
 - Dãy con lớn nhất của dãy a_1, a_2, \dots, a_i , và dãy con này kết thúc tại phần tử a_i .

→ Do đó, ta có $s_i = \max \{s_{i-1}, e_i\}, i = 2, \dots, n$.

với e_i là trọng lượng của dãy con lớn nhất a_1, a_2, \dots, a_i và dãy con này kết thúc tại a_i .

Để tính e_i , ta xây dựng công thức đệ quy:

- $e_1 = a_1$;
- $e_i = \max \{a_i, e_{i-1} + a_i\}, i = 2, \dots, n$.

1.1.4. Thuật toán quy hoạch động giải bài toán dãy con lớn nhất

Thuật toán quy hoạch động được chia làm 3 giai đoạn:

1. Phân rã:

- Gọi s_i là trọng lượng của dãy con lớn nhất của dãy $a_1, a_2, \dots, a_i, i = 1, 2, \dots, n$.
- Rõ ràng, s_n là giá trị cần tìm (lời giải của bài toán).

3. Tổng hợp lời giải:

- $s_1 = a_1$
- Giả sử $i > 1$ và ta đã biết giá trị s_k với $k = 1, 2, \dots, i-1$. Ta cần tính giá trị s_i là trọng lượng của dãy con lớn nhất của dãy:

$$a_1, a_2, \dots, a_{i-1}, a_i.$$

- Nhận thấy rằng: dãy con lớn nhất của dãy $a_1, a_2, \dots, a_{i-1}, a_i$ có thể hoặc bao gồm phần tử a_i hoặc không bao gồm phần tử a_i → do đó, dãy con lớn nhất của dãy $a_1, a_2, \dots, a_{i-1}, a_i$ chỉ có thể là một trong 2 dãy sau:
 - Dãy con lớn nhất của dãy a_1, a_2, \dots, a_{i-1}
 - Dãy con lớn nhất của dãy a_1, a_2, \dots, a_i , và dãy con này kết thúc tại phần tử a_i .

→ Do đó, ta có $s_i = \max \{s_{i-1}, e_i\}, i = 2, \dots, n$.

với e_i là trọng lượng của dãy con lớn nhất a_1, a_2, \dots, a_i và dãy con này kết thúc tại a_i .

Để tính e_i , ta xây dựng công thức đệ quy:

- $e_1 = a_1$;
- $e_i = \max \{a_i, e_{i-1} + a_i\}, i = 2, \dots, n$.

```
MaxSub(a)
{
    smax = a[1];           // smax : trọng lượng của dãy con lớn nhất
    ei   = a[1];           // ei: trọng lượng của dãy con lớn nhất kết thúc tại phần tử a[i]
    for i = 2 to n {
        ei = max {a[i], ei + a[i] };
        smax = max {smax, ei};
    }
}
```

1.1.4. Thuật toán quy hoạch động giải bài toán dãy con lớn nhất

MaxSub(a)

```
{
    smax = a[1];           // smax : trọng lượng của dãy con lớn nhất
    ei    = a[1];           // ei: trọng lượng của dãy con lớn nhất kết thúc tại phần tử a[i]
    imax = 1;               // imax : chỉ số của phần tử cuối cùng thuộc dãy con lớn nhất
    for i = 2 to n {
        u = ei + a[i];
        v = a[i];
        if (u > v) ei = u
        else ei = v;
        if (ei > smax) {
            smax := ei;
            imax := i;
        }
    }
}
```

MaxSub(a)

```
{
    smax = a[1];           // smax : trọng lượng của dãy con lớn nhất
    ei    = a[1];           // ei: trọng lượng của dãy con lớn nhất kết thúc tại phần tử a[i]
    for i = 2 to n {
        ei = max {a[i], ei + a[i]};
        smax = max {smax, ei};
    }
}
```

Phân tích thuật toán:

Số phép cộng phải thực hiện trong thuật toán

= Số lần thực hiện câu lệnh **u = ei + a[i]**

= n

So sánh 4 thuật toán

- Bảng sau cho thấy ước tính thời gian tính của 4 thuật toán đề xuất ở trên (với giả thiết máy tính có thể thực hiện 10^8 phép cộng trong 1 giây)

Thuật toán	Độ phức tạp	$n=10^4$	Thời gian	$n=10^6$	Thời gian
Duyệt toàn bộ	n^3	10^{12}	2.7 giờ	10^{18}	115 ngày
Duyệt toàn bộ có cải tiến	n^2	10^8	1 giây	10^{12}	2.7 giờ
Đệ quy (Recursive)	$n \log n$	$1.33 \cdot 10^5$	10^{-3} giây	$1.99 \cdot 10^7$	$2 \cdot 10^{-1}$ giây
Quy hoạch động (Dynamic programming)	n	10^4	10^{-4} giây	10^6	$2 \cdot 10^{-2}$ giây

- Ví dụ này cho ta thấy việc phát triển được thuật toán hiệu quả có thể giảm bớt được chi phí thời gian một cách đáng kể như thế nào.

Nội dung

1.1. Ví dụ mở đầu

1.2. Thuật toán và độ phức tạp

1.3. Kí hiệu tiệm cận

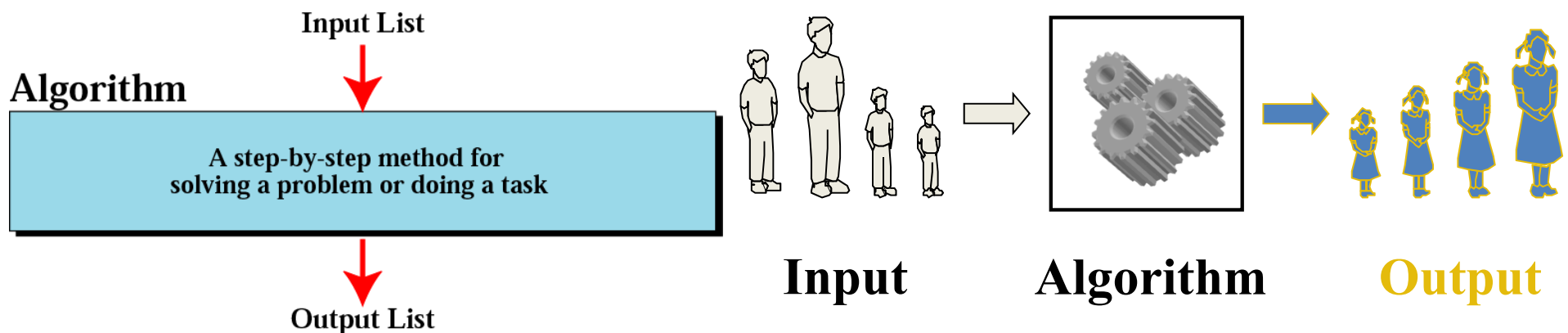
1.4. Giả ngôn ngữ (Pseudo code)

1.5. Một số kĩ thuật phân tích thuật toán

1.6. Giải công thức đệ quy

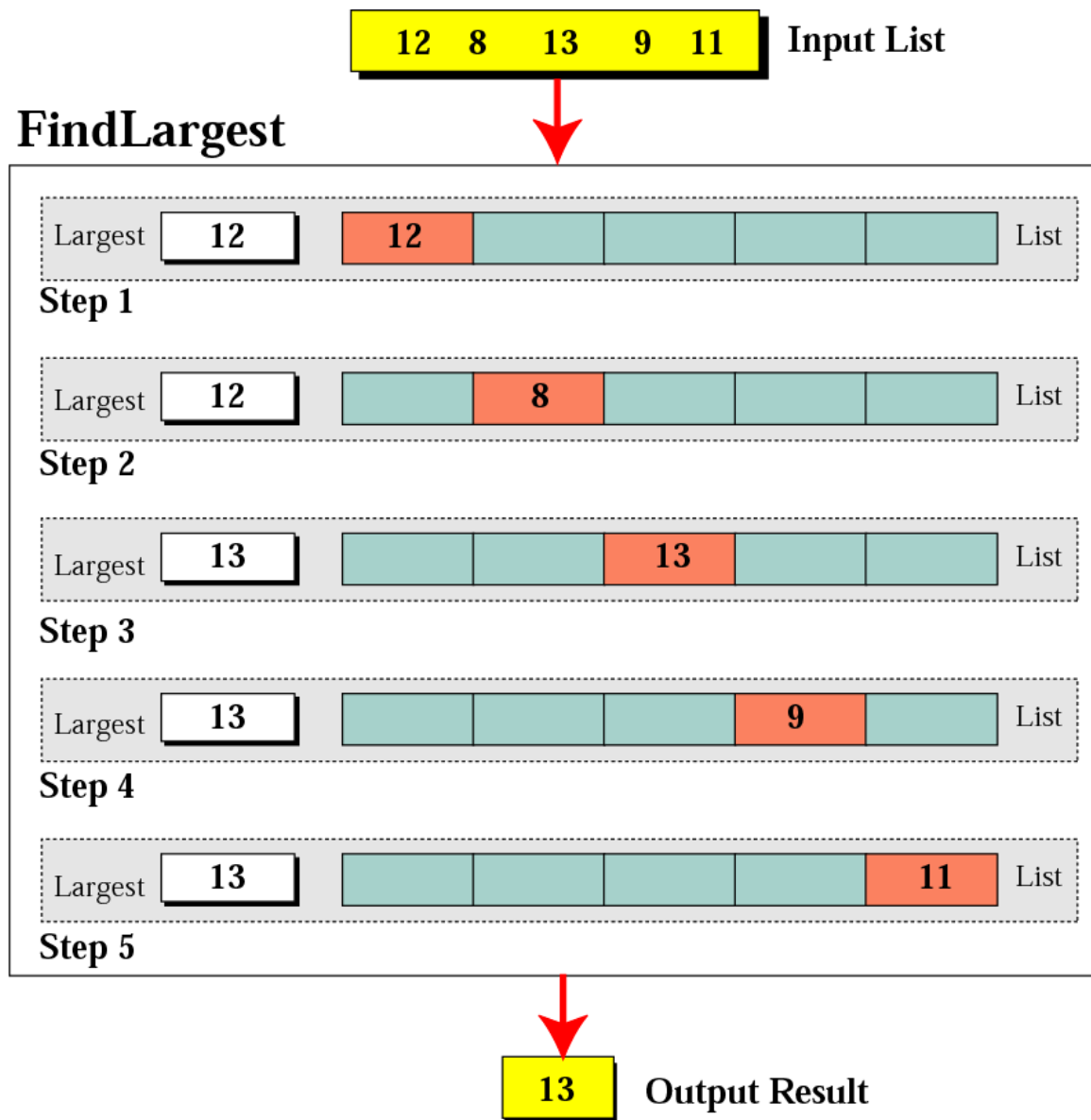
Thuật toán (Algorithm)

- Thuật ngữ *algorithm* xuất phát từ tên của nhà toán học người Ba Tư: Abu Ja'far Mohammed ibn-i Musa al Khowarizmi.
- Trong ngành khoa học máy tính, thuật ngữ “thuật toán” được dùng để chỉ một phương pháp bao gồm **một chuỗi các câu lệnh** mà máy tính có thể sử dụng để giải quyết một bài toán.
- Định nghĩa.** Ta hiểu thuật toán giải bài toán đặt ra là một thủ tục xác định bao gồm một dãy hữu hạn các bước cần thực hiện để **thu được đầu ra (output)** cho **một đầu vào cho trước (input)** của bài toán.



- Ví dụ: Bài toán tìm số nguyên lớn nhất trong một dãy các số nguyên dương cho trước
 - Đầu vào (Input) : dãy gồm n số nguyên dương a_1, a_2, \dots, a_n
 - Đầu ra (Output): số có giá trị lớn nhất của dãy đã cho
 - Ví dụ: Input 12 8 13 9 11 \rightarrow Output: 12
 - Yêu cầu: thiết kế thuật toán giải bài toán trên

Ví dụ: Tìm số lớn nhất trong dãy số gồm 5 số nguyên sau:



Các bước trong thuật toán FindLargest

12 8 13 9 11 Input List

FindLargest

Đặt Largest = phần tử đầu tiên của dãy

Step 1

Nếu phần tử thứ hai lớn hơn Largest: đặt Largest = phần tử thứ hai của dãy

Step 2

Nếu phần tử thứ ba lớn hơn Largest: đặt Largest = phần tử thứ ba của dãy

Step 3

Nếu phần tử thứ tư lớn hơn Largest: đặt Largest = phần tử thứ tư của dãy

Step 4

Nếu phần tử thứ năm lớn hơn Largest: đặt Largest = phần tử thứ năm của dãy

Step 5

13

Output Result

FindLargest chỉnh sửa cho hợp lý

12 8 13 9 11 Input List

FindLargest

Đặt Largest = 0

Step 0

Nếu số hiện tại lớn hơn Largest, đặt Largest = số hiện tại

Step 1

⋮

Nếu số hiện tại lớn hơn Largest, đặt Largest = số hiện tại

Step 5

13

Output Result

Tổng quát hàm FindLargest

Input List



FindLargest

Đặt Largest = 0

Repeat the following step N times:

Nếu số hiện tại lớn hơn Largest, đặt Largest = số hiện tại



Largest

Thuật toán

- Thuật toán có các đặc trưng sau đây:
 - **Đầu vào (Input):** Thuật toán nhận dữ liệu vào từ một tập nào đó.
 - **Đầu ra (Output):** Với mỗi tập các dữ liệu đầu vào, thuật toán đưa ra các dữ liệu tương ứng với lời giải của bài toán.
 - **Chính xác (Precision):** Các bước của thuật toán được mô tả chính xác.
 - **Hữu hạn (Finiteness):** Thuật toán cần phải đưa được đầu ra sau một số hữu hạn (có thể rất lớn) bước với mọi đầu vào.
 - **Đơn trị (Uniqueness):** Các kết quả trung gian của từng bước thực hiện thuật toán được xác định một cách đơn trị và chỉ phụ thuộc vào đầu vào và các kết quả của các bước trước.
 - **Tổng quát (Generality):** Thuật toán có thể áp dụng để giải mọi bài toán có dạng đã cho.

Ví dụ: Xây dựng thuật toán giải bài toán

- Bài toán: Thiết kế chương trình sắp xếp $n \geq 1$ số nguyên theo thứ tự không giảm
 - Input: Tập n số nguyên: $a[1], a[2], \dots, a[n]$
 - Output: Tập n số nguyên đã được sắp xếp: $a[1] \leq a[2] \leq \dots \leq a[n]$
- Bước I – Phân tích
 - Từ dãy số chưa được sắp xếp, ta tìm số nhỏ nhất và đặt số này vào vị trí tiếp theo trong dãy đã sắp xếp
- Bước II – Thuật toán

```
for (i = 1; i <= n; i++) {
  - Trong dãy các phần tử từ  $a[i]$  đến  $a[n]$  tìm phần tử có giá trị nhỏ nhất, và giả sử số nguyên nhỏ nhất tìm được là  $a[\text{index\_min}]$  với  $i \leq \text{index\_min} \leq n$ ;
  - Hoán đổi vị trí  $a[i]$  và  $a[\text{index\_min}]$ ;}
```

- Bước III – Cài đặt

```
void sort(int *a, int n)
{
    for (i= 1; i < n; i++)
    {
        int index_min= i;
        for (int k= i+1; k<= n; k++)
            if (a[k] < a[index_min]) index_min= k;
        int temp=a[i]; a[i]=a[index_min]; a[index_min]=temp;
    }
}
```

Ví dụ: Mảng gồm 6 số nguyên

a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
9	6	-7	8	10	3

Độ phức tạp của thuật toán

Cho 2 thuật toán cùng giải một bài toán cho trước, làm thế nào để xác định thuật toán nào tốt hơn?

- Khi nói đến hiệu quả của một thuật toán, ta quan tâm đến chi phí cần dùng để thực hiện nó:
 - 1) Dễ hiểu, dễ cài đặt, dễ sửa đổi?
 - 2) Thời gian sử dụng CPU? **THỜI GIAN**
 - 3) Tài nguyên bộ nhớ? **BỘ NHỚ**

Trong giáo trình này ta đặc biệt quan tâm đến đánh giá thời gian cần thiết để thực hiện thuật toán mà ta sẽ gọi là *thời gian tính* của thuật toán.

Làm thế nào để đo được thời gian tính

- Mọi thuật toán đều thực hiện chuyển đổi đầu vào thành đầu ra:



- Thời gian tính của thuật toán phụ thuộc vào dữ liệu vào (kích thước tăng, thì thời gian tăng).
 - Định nghĩa.** *Ta gọi kích thước dữ liệu đầu vào (hay độ dài dữ liệu vào) là số bit cần thiết để biểu diễn nó.*
 - Vậy, ta sẽ tìm cách đánh giá thời gian tính của thuật toán bởi một hàm của độ dài dữ liệu đầu vào.
 - Tuy nhiên, trong một số trường hợp, kích thước dữ liệu đầu vào là như nhau, nhưng thời gian tính lại rất khác nhau
 - Ví dụ: Để tìm số nguyên tố đầu tiên có trong dãy: ta duyệt dãy từ trái sang phải
Dãy 1: 3 9 8 12 15 20 (thuật toán dừng ngay khi xét phần tử đầu tiên)
Dãy 2: 9 8 3 12 15 20 (thuật toán dừng khi xét phần tử thứ ba)
Dãy 3: 9 8 12 15 20 3 (thuật toán dừng khi xét phần tử cuối cùng)
→ 3 loại thời gian tính

Các loại thời gian tính

Thời gian tính tốt nhất (Best-case):

- $T(n)$: Thời gian tối thiểu cần thiết để thực hiện thuật toán với mọi bộ dữ liệu đầu vào kích thước n .
- Dùng với các thuật toán chậm chạy nhanh với một vài đầu vào.

Thời gian tính trung bình (Average-case):

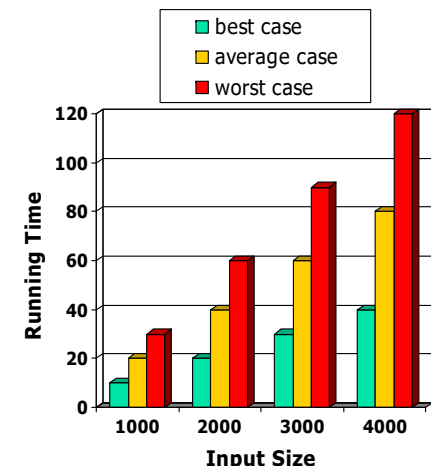
- $T(n)$: Thời gian trung bình cần thiết để thực hiện thuật toán trên tập hữu hạn các đầu vào kích thước n .
- Rất hữu ích, nhưng khó xác định

Thời gian tính tồi nhất (Worst-case):

- $T(n)$: Thời gian nhiều nhất cần thiết để thực hiện thuật toán với mọi bộ dữ liệu đầu vào kích thước n . Thời gian như vậy sẽ được gọi là **thời gian tính tồi nhất** của thuật toán với đầu vào kích thước n .
- Dễ xác định

Có hai cách để đánh giá thời gian tính:

- Từ thời gian chạy thực nghiệm
- Lý thuyết: khái niệm xấp xỉ tiệm cận



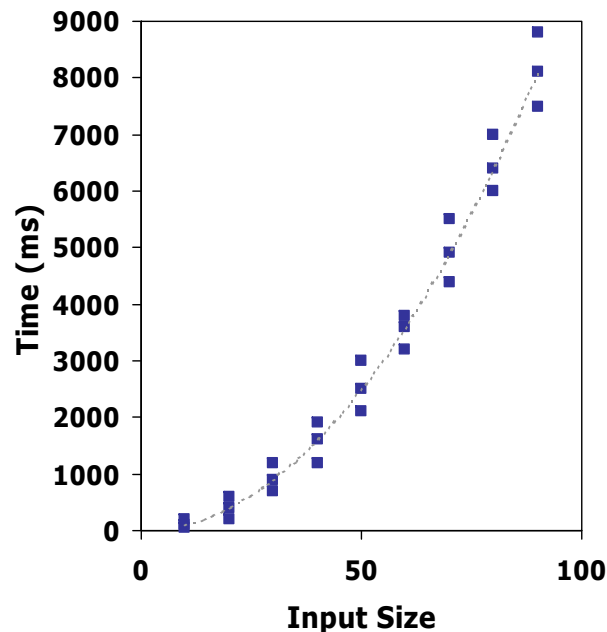
Phân tích thời gian tính của thuật toán bằng thời gian chạy thực nghiệm

Ta có thể đánh giá thuật toán bằng phương pháp thực nghiệm thông qua việc cài đặt thuật toán, rồi chọn các bộ dữ liệu đầu vào thử nghiệm:

- Viết chương trình thực hiện thuật toán
- Chạy chương trình với các dữ liệu đầu vào kích thước khác nhau
- Sử dụng hàm `clock ()` để đo thời gian chạy chương trình

```
clock_t startTime = clock();  
doSomeOperation();  
clock_t endTime = clock();  
clock_t clockTicksTaken = endTime - startTime;  
double timeInSeconds = clockTicksTaken / (double) CLOCKS_PER_SEC;
```

- Vẽ đồ thị kết quả



Nhược điểm của việc đánh giá thời gian tính của thuật toán dựa vào thời gian chạy thực nghiệm chương trình

- Bắt buộc phải cài đặt chương trình thì mới có thể đánh giá được thời gian tính của thuật toán. Do phải cài đặt bằng một ngôn ngữ lập trình cụ thể nên thuật toán sẽ chịu sự hạn chế của ngữ lập trình này. Đồng thời, hiệu quả của thuật toán sẽ bị ảnh hưởng bởi trình độ của người cài đặt.
 - Kết quả thu được sẽ không bao gồm thời gian chạy của những dữ liệu đầu vào không được chạy thực nghiệm. Do vậy, cần chọn được các bộ dữ liệu thử đặc trưng cho tất cả tập các dữ liệu vào của thuật toán → rất khó khăn và tốn nhiều chi phí.
 - Để so sánh thời gian tính của hai thuật toán, cần phải chạy thực nghiệm hai thuật toán trên cùng một máy, và sử dụng cùng một phần mềm.
- Do đó người ta đã tìm kiếm những phương pháp đánh giá thuật toán hình thức hơn, ít phụ thuộc môi trường cũng như phần cứng hơn. Một phương pháp như vậy là phương pháp đánh giá thuật toán theo hướng xấp xỉ tiệm cận

Phân tích lý thuyết thời gian tính của thuật toán

- Sử dụng giả ngôn ngữ (pseudo-code) để mô tả thuật toán, thay vì tiến hành cài đặt thực sự
- Phân tích thời gian tính của thuật toán như là hàm của kích thước dữ liệu đầu vào, n
- Phân tích tất cả các trường hợp có thể có của dữ liệu đầu vào
- Cho phép ta có thể đánh giá thời gian tính của thuật toán không bị phụ thuộc vào phần cứng/phần mềm chạy chương trình (Thay đổi phần cứng/phần mềm chỉ làm thay đổi thời gian chạy một lượng hằng số, chứ không làm thay đổi tốc độ tăng của thời gian tính)

Nội dung

1.1. Ví dụ mở đầu

1.2. Thuật toán và độ phức tạp

1.3. Kí hiệu tiệm cận

1.4. Giả ngôn ngữ (Pseudo code)

1.5. Một số kĩ thuật phân tích thuật toán

1.6. Giải công thức đệ quy

1.3. Kí hiệu tiệm cận (Asymptotic notation)

Θ , Ω , O , o , ω

» Những kí hiệu này:

- Được sử dụng để mô tả thời gian tính của thuật toán, mô tả tốc độ tăng của thời gian chạy phụ thuộc vào kích thước dữ liệu đầu vào.
- Được xác định đối với các hàm nhận giá trị nguyên không âm
- Dùng để so sánh tốc độ tăng của 2 hàm

Ví dụ: $f(n) = \Theta(n^2)$: mô tả tốc độ tăng của hàm $f(n)$ và n^2 .

» Thay vì cố gắng tìm ra một công thức phức tạp để tính chính xác thời gian tính của thuật toán, ta nói thời gian tính cỡ $\Theta(n^2)$ [đọc là theta n^2]: tức là, thời gian tính tỉ lệ thuận với n^2 cộng thêm các đa thức bậc thấp hơn

Θ – Kí hiệu Theta

Đối với hàm $g(n)$ cho trước, ta kí hiệu $\Theta(g(n))$ là tập các hàm

$\Theta(g(n)) = \{f(n): \text{tồn tại các hằng số } c_1, c_2 \text{ và } n_0 \text{ sao cho}$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \text{ với mọi } n \geq n_0 \}$$

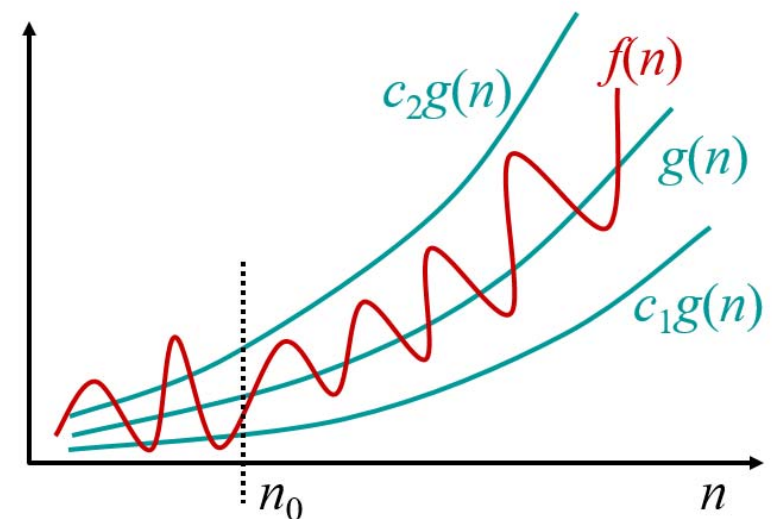
(tập tất cả các hàm có cùng tốc độ tăng với hàm $g(n)$).

- Hàm $f(n)$ thuộc vào tập $\Theta(g(n))$ nếu tồn tại hằng số dương c_1 và c_2 sao cho nó bị “kẹp” giữa $c_1 g(n)$ và $c_2 g(n)$ với giá trị n đủ lớn
 - $f(n) = \Theta(g(n))$ tức là tồn tại hằng số c_1 and c_2 sao cho
$$c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ với giá trị } n \text{ đủ lớn.}$$

Ta nói $g(n)$ là đánh giá tiệm cận đúng cho $f(n)$

và viết $f(n) = \Theta(g(n))$

- Khi ta nói một hàm là theta của hàm khác, nghĩa là không có hàm nào đạt tới giá trị vô cùng nhanh hơn



$$f(n) = \Theta(g(n)) \iff \exists c_1, c_2, n_0 > 0 : \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Ví dụ 1: Chứng minh rằng $10n^2 - 3n = \Theta(n^2)$

- Ta cần chỉ ra với những giá trị nào n_0, c_1, c_2 thì bất đẳng thức trong định nghĩa của kí hiệu theta là đúng:

$$c_1 n^2 \leq f(n) = 10n^2 - 3n \leq c_2 n^2 \quad \forall n \geq n_0$$

- Gợi ý: lấy c_1 nhỏ hơn hệ số của số hạng với số mũ cao nhất, và c_2 lấy lớn hơn.

→ Chọn: $c_1 = 1, c_2 = 11, n_0 = 1$ thì ta có

$$n^2 \leq 10n^2 - 3n \leq 11n^2, \text{ với } n \geq 1.$$

→ $\forall n \geq 1: 10n^2 - 3n = \Theta(n^2)$

- Chú ý: Với các hàm đa thức: để so sánh tốc độ tăng, ta cần nhìn vào số hạng có số mũ cao nhất

$$f(n) = \Theta(g(n)) \iff \exists c_1, c_2, n_0 > 0 : \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Ví dụ 2: Chứng minh rằng $f(n) = \frac{1}{2}n^2 - 3n = \Theta(n^2)$

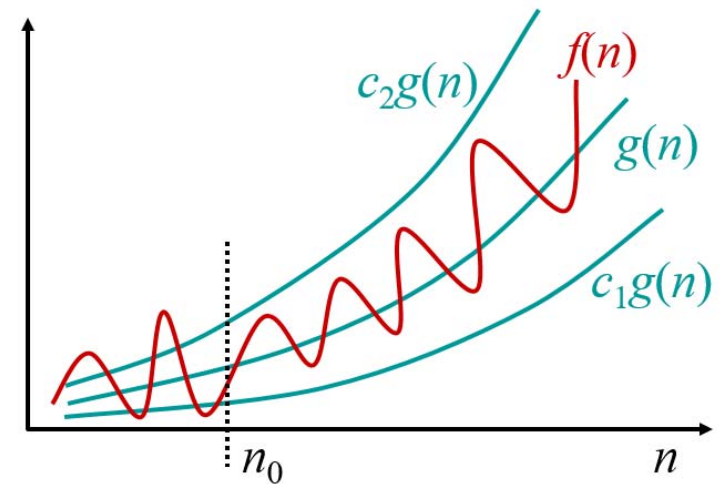
Ta cần tìm n_0 , c_1 và c_2 sao cho

$$c_1 n^2 \leq f(n) = \frac{1}{2}n^2 - 3n \leq c_2 n^2 \quad \forall n \geq n_0$$

Chọn $c_1 = 1/4$, $c_2 = 1$, và $n_0 = 7$ ta có:

$$\frac{1}{4}n^2 \leq f(n) = \frac{1}{2}n^2 - 3n \leq n^2 \quad \forall n \geq 7$$

$$\Rightarrow \forall n \geq 7: \quad \frac{1}{2}n^2 - 3n = \Theta(n^2)$$



$$f(n) = \Theta(g(n))$$

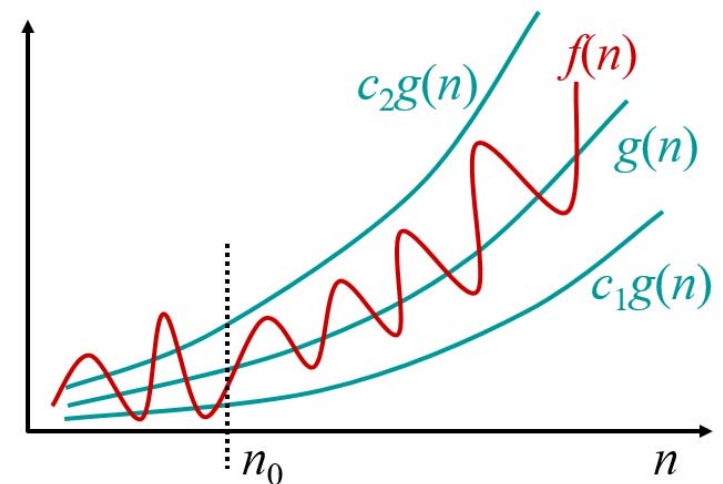


$$\exists c_1, c_2, n_0 > 0 : \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Ví dụ 3: Chứng minh rằng $f(n) = 23n^3 - 10n^2 \log_2 n + 7n + 6 = \Theta(n^3)$

Ta phải tìm n_0 , c_1 và c_2 sao cho

$$c_1 n^3 \leq f(n) = 23n^3 - 10n^2 \log_2 n + 7n + 6 \leq c_2 n^3 \text{ với } \forall n \geq n_0$$



$$f(n) = \Theta(g(n))$$



$$\exists c_1, c_2, n_0 > 0 : \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Ví dụ 3: Chứng minh rằng $f(n) = 23n^3 - 10n^2 \log_2 n + 7n + 6 = \Theta(n^3)$

$$f(n) = 23n^3 - 10n^2 \log_2 n + 7n + 6$$

$$\rightarrow f(n) = [23 - (10 \log_2 n)/n + 7/n^2 + 6/n^3]n^3$$

Khi đó:

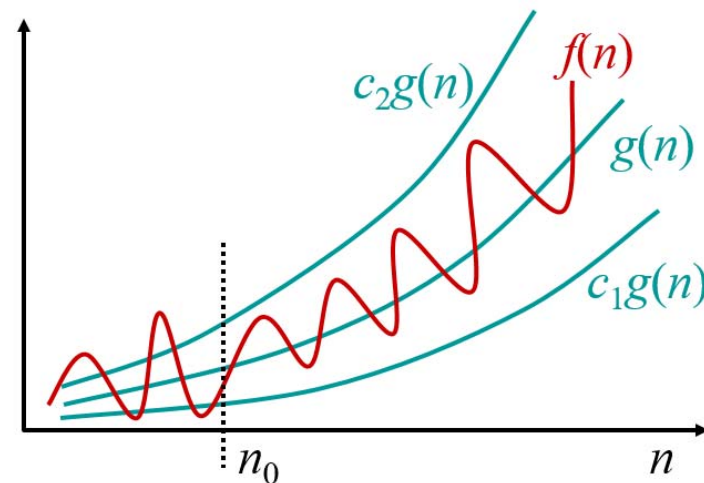
- $\forall n \geq 10 : f(n) \leq (23 + 0 + 7/100 + 6/1000)n^3$
 $= (23 + 0 + 0.07 + 0.006)n^3$
 $= 23.076 n^3 < 24 n^3$
- $\forall n \geq 10 : f(n) \geq (23 - \log_2 10 + 0 + 0)n^3 > (23 - \log_2 16)n^3 = 19n^3$

\rightarrow Ta có:

$$\forall n \geq 10: 19 n^3 \leq f(n) \leq 24 n^3$$

$$(n_0 = 10, c_1 = 19, c_2 = 24, g(n) = n^3)$$

$$\text{Do đó: } f(n) = \Theta(n^3)$$



O – Kí hiệu O lớn (big Oh)

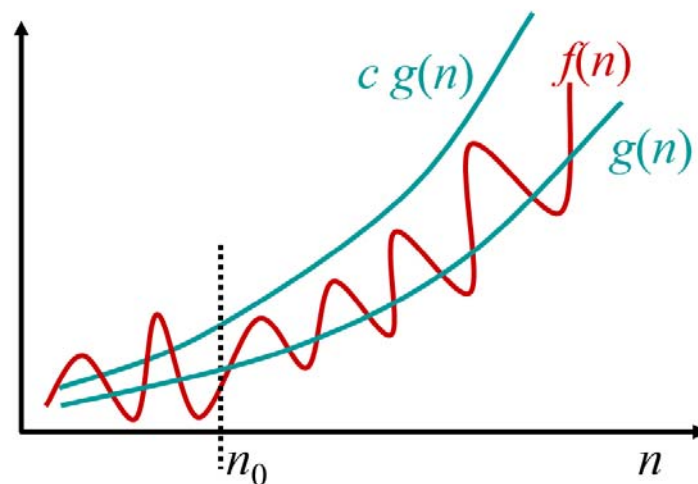
Đối với hàm $g(n)$ cho trước, ta ký hiệu $O(g(n))$ là tập các hàm

$O(g(n)) = \{f(n): \text{tồn tại các hằng số dương } c \text{ và } n_0 \text{ sao cho:}$

$$f(n) \leq cg(n) \text{ với mọi } n \geq n_0 \}$$

(tập tất cả các hàm có **tốc độ tăng nhỏ hơn hoặc bằng** tốc độ tăng của $g(n)$).

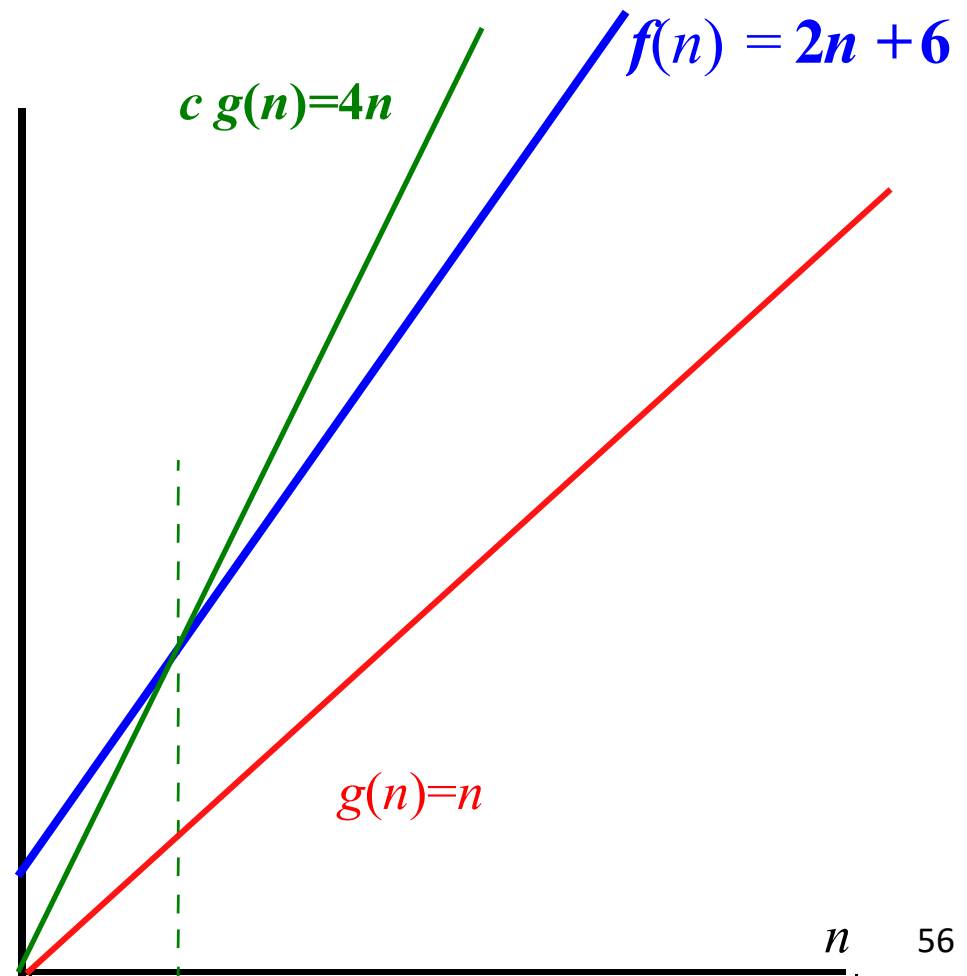
- Ta nói: $g(n)$ là cận trên tiệm cận của $f(n)$, và viết $f(n) = O(g(n))$.
- $f(n) = O(g(n))$ tức là tồn tại hằng số c sao cho $f(n)$ luôn $\leq cg(n)$ với mọi giá trị n đủ lớn.
- $O(g(n))$ là tập các hàm đạt tới giá trị vô cùng không nhanh hơn $g(n)$.



Minh họa hình học

$$O(g(n)) = \{f(n) : \exists \text{ hằng số dương } c \text{ và } n_0, \text{ sao cho} \\ \forall n \geq n_0, \text{ sao cho } 0 \leq f(n) \leq cg(n)\}$$

- $f(n) = 2n+6$
 - Theo định nghĩa:
 - Cần tìm hàm $g(n)$ và hằng số c và n_0 sao cho $f(n) < cg(n)$ khi $n > n_0$
- $g(n) = n, c = 4$ và $n_0=3$
- $f(n)$ là $O(n)$



Ví dụ O lớn (Big-Oh)

$$O(g(n)) = \{f(n) : \exists \text{ hằng số dương } c \text{ và } n_0, \text{ sao cho} \\ \forall n \geq n_0, \text{ ta có } 0 \leq f(n) \leq cg(n) \}$$

- Ví dụ 1: Chứng minh rằng $2n + 10 = O(n)$

→ $f(n) = 2n+10, g(n) = n$

- Cần tìm hằng số c và n_0 sao cho $2n + 10 \leq cn$ với mọi $n \geq n_0$
- $(c - 2) n \geq 10$
- $n \geq 10/(c - 2)$
- Chọn $c = 3$ và $n_0 = 10$

- Ví dụ 2: Chứng minh rằng $7n - 2 = O(n)$

→ $f(n) = 7n-2, g(n) = n$

- Cần tìm hằng số c và n_0 sao cho $7n - 2 \leq cn$ với mọi $n \geq n_0$
- $(7 - c) n \leq 2$
- $n \leq 2/(7 - c)$
- Chọn $c = 7$ và $n_0 = 1$

Chú ý

- Các giá trị của các hằng số dương n_0 và c **không phải là duy nhất** trong chứng minh công thức tiệm cận
- Ví dụ: Chứng minh rằng $100n + 5 = O(n^2)$
 - $100n + 5 \leq 100n + n = 101n \leq 101n^2 \quad \forall n \geq 5$
 $n_0 = 5$ and $c = 101$ là các hằng số cần tìm
 - $100n + 5 \leq 100n + 5n = 105n \leq 105n^2 \quad \forall n \geq 1$
 $n_0 = 1$ and $c = 105$ cũng là các hằng số cần tìm
- Chỉ cần tìm các hằng số dương c và n_0 **nào đó** thỏa mãn bất đẳng thức trong định nghĩa công thức tiệm cận

Ví dụ O lớn (Big-Oh)

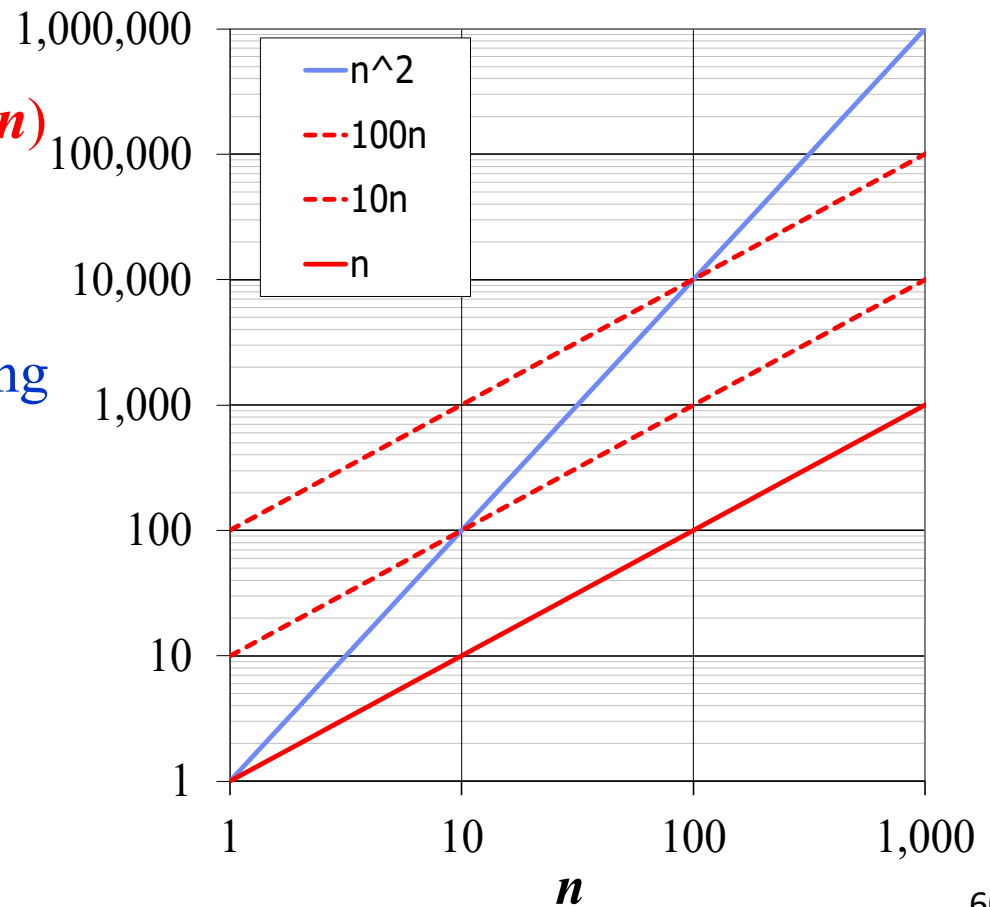
$$O(g(n)) = \{f(n) : \exists \text{ hằng số dương } c \text{ và } n_0, \text{ sao cho} \\ \forall n \geq n_0, \text{ ta có } 0 \leq f(n) \leq cg(n) \}$$

- Ví dụ 3: Chứng minh rằng $3n^3 + 20n^2 + 5 = O(n^3)$
Cần tìm giá trị cho c và n_0 sao cho $3n^3 + 20n^2 + 5 \leq cn^3$ với mọi $n \geq n_0$
Bất đẳng thức đúng với $c = 4$ và $n_0 = 21$
- Ví dụ 4: Chứng minh rằng $3 \log_2 n + 5 = O(\log_2 n)$

Ví dụ O lớn (Big-Oh)

$$O(g(n)) = \{f(n) : \exists \text{ hằng số dương } c \text{ và } n_0, \text{ sao cho} \\ \forall n \geq n_0, \text{ ta có } 0 \leq f(n) \leq cg(n)\}$$

- Ví dụ 5: hàm n^2 không thuộc $O(n)$
 - $n^2 \leq cn$
 - $n \leq c$
 - Bất đẳng thức trên không đúng vì c phải là hằng số



O lớn và tốc độ tăng

- Kí hiệu O lớn cho cận trên tốc độ tăng của một hàm
- Khi ta nói “ $f(n)$ là $O(g(n))$ ” nghĩa là tốc độ tăng của hàm $f(n)$ không lớn hơn tốc độ tăng của hàm $g(n)$
- Ta có thể dùng kí hiệu O lớn để sắp xếp các hàm theo tốc độ tăng của chúng

	$f(n)$ là $O(g(n))$	$g(n)$ là $O(f(n))$
$g(n)$ có tốc độ tăng lớn hơn $f(n)$	Yes	No
$g(n)$ có tốc độ tăng nhỏ hơn $f(n)$?	?
$g(n)$ và $f(n)$ cùng tốc độ tăng	?	?

Các biểu thức sai

$$f(n) \not\leq O(g(n))$$

$$f(n) \not\geq O(g(n))$$

Ví dụ O lớn (Big-Oh)

- $f(n) = 50n^3 + 20n + 4$ là $O(n^3)$
 - Cũng đúng khi nói $f(n)$ là $O(n^3+n)$
 - Không hữu ích, vì n^3 có tốc độ tăng lớn hơn rất nhiều so với n , khi n lớn
 - Cũng đúng khi nói $f(n)$ là $O(n^5)$
 - OK, nhưng $g(n)$ nên có tốc độ tăng càng gần với tốc độ tăng của $f(n)$ càng tốt, thì đánh giá thời gian tính mới có giá trị
- $3\log(n) + \log(\log(n)) = O(?)$

• **Quy tắc đơn giản:** Bỏ qua các số hạng có số mũ thấp hơn và các hằng số

Một số quy tắc O lớn

- Nếu $f(n)$ là đa thức bậc d :
$$f(n) = a_0 + a_1n + a_2n^2 + \dots + a_dn^d$$

thì $f(n) = O(n^d)$, tức là,

- Bỏ qua các số hạng có số mũ thấp hơn d
- Bỏ qua các hằng số

Ví dụ: $3n^3 + 20n^2 + 5 = O(n^3)$

- Nếu $f(n) = O(n^k)$ thì $f(n) = O(n^p)$ với $\forall p > k$

Ví dụ: $2n^2 = O(n^2)$ thì $2n^2 = O(n^3)$

Khi đánh giá tiệm cận $f(n) = O(g(n))$, ta nên tìm hàm $g(n)$ có tốc độ tăng càng chậm càng tốt

- Sử dụng lớp các hàm có tốc độ tăng nhỏ nhất có thể

Ví dụ: Nói “ $2n = O(n)$ ” tốt hơn là nói “ $2n = O(n^2)$ ”

- Sử dụng lớp các hàm đơn giản nhất có thể

Ví dụ: Nói “ $3n + 5 = O(n)$ ” thay vì nói “ $3n + 5 = O(3n)$ ”

Ví dụ O lớn

- Tất cả các hàm sau đều là $O(n)$:
 - $n, 3n, 61n + 5, 22n - 5, \dots$
- Tất cả các hàm sau đều là $O(n^2)$:
 - $n^2, 9n^2, 18n^2 + 4n - 53, \dots$
- Tất cả các hàm sau đều là $O(n \log n)$:
 - $n(\log n), 5n(\log 99n), 18 + (4n - 2)(\log (5n + 3)), \dots$

Ω - kí hiệu Omega

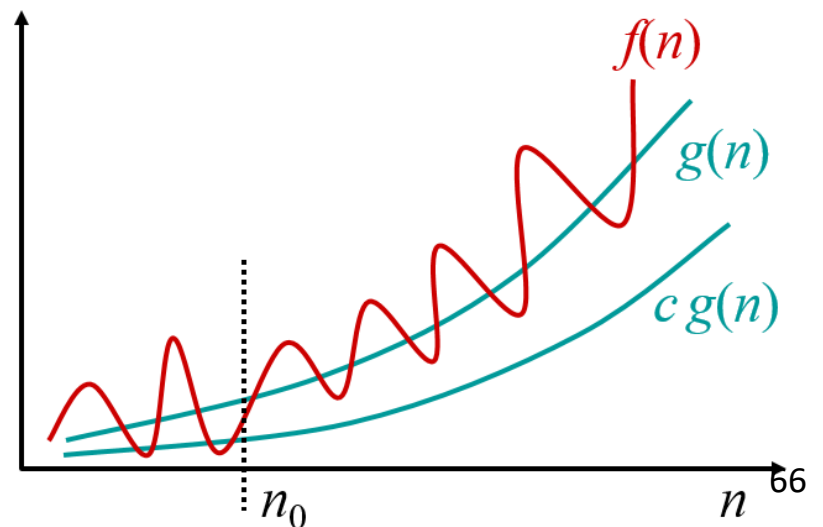
- Đối với hàm $g(n)$ cho trước, ta kí hiệu $\Omega(g(n))$ là tập các hàm

$$\Omega(g(n)) = \{f(n): \text{tồn tại các hằng số dương } c \text{ và } n_0 \text{ sao cho:}$$
$$cg(n) \leq f(n) \text{ với mọi } n \geq n_0 \}$$

(tập tất cả các hàm có **tốc độ tăng lớn hơn hoặc bằng** tốc độ tăng của $g(n)$).

- Ta nói: $g(n)$ là cận dưới tiệm cận của hàm, và viết $f(n) = \Omega(g(n))$.
- $f(n) = \Omega(g(n))$ nghĩa là tồn tại hằng số c sao cho $f(n)$ luôn $\geq cg(n)$ với giá trị n đủ lớn.
- $\Omega(g(n))$ là tập các hàm đạt tới giá trị vô cùng không chậm hơn $g(n)$

$$f(n) = \Theta(g(n)) \Rightarrow f(n) = \Omega(g(n))$$
$$\Theta(g(n)) \subset \Omega(g(n)).$$



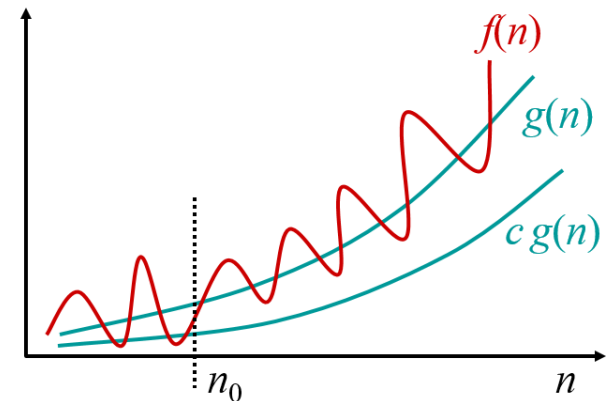
Ví dụ Kí hiệu Omega

$$\Omega(g(n)) = \{f(n) : \exists \text{ hằng số dương } c \text{ và } n_0, \text{ sao cho} \\ \forall n \geq n_0, \text{ ta có } 0 \leq cg(n) \leq f(n)\}$$

- Ví dụ 1: Chứng minh rằng $5n^2 = \Omega(n)$
Cần tìm c và n_0 sao cho $cn \leq 5n^2$ với $n \geq n_0$
Bất đẳng thức đúng với $c = 1$ và $n_0 = 1$

Nhận xét:

- Nếu $f(n) = \Omega(n^k)$ thì $f(n) = \Omega(n^p)$ với $\forall p < k$.
- Khi đánh giá tiệm cận $f(n) = \Omega(g(n))$, ta cần tìm hàm $g(n)$ có tốc độ tăng càng nhanh càng tốt
- Ví dụ 2: Chứng minh $\sqrt{n} = \Omega(\lg n)$



Kí hiệu tiệm cận trong các đẳng thức

Kí hiệu tiệm cận còn được dùng để thay thế các biểu thức chứa các toán hạng với tốc độ tăng chậm.

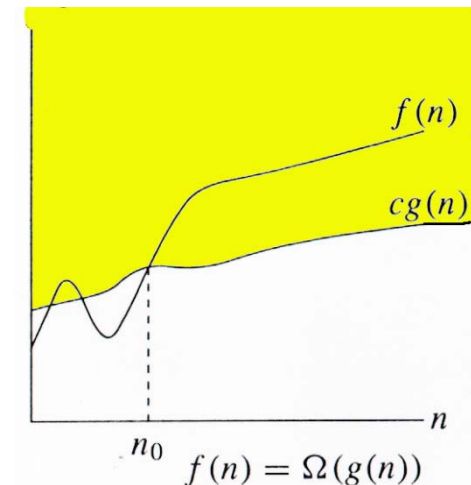
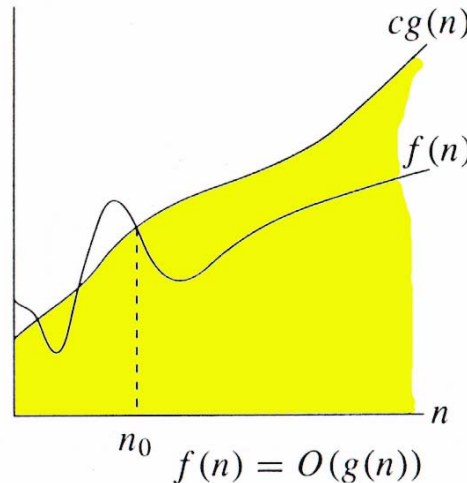
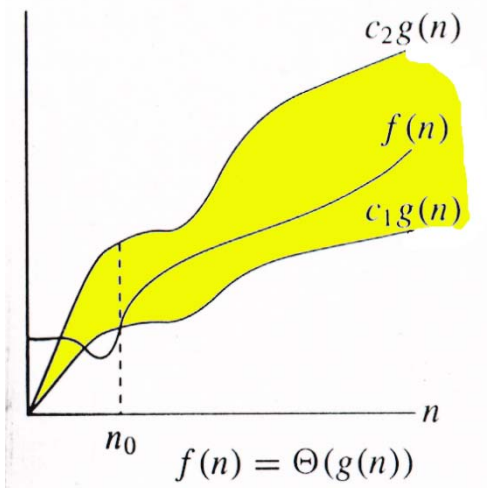
Ví dụ:

$$\begin{aligned}4n^3 + 3n^2 + 2n + 1 &= 4n^3 + 3n^2 + \Theta(n) \\ &= 4n^3 + \Theta(n^2) = \Theta(n^3)\end{aligned}$$

Trong các đẳng thức, $\Theta(f(n))$ thay thế cho một hàm nào đó $g(n) \in \Theta(f(n))$

– Trong ví dụ trên, ta dùng $\Theta(n^2)$ thay thế cho $3n^2 + 2n + 1$

Các kí hiệu tiệm cận



Đồ thị minh họa cho các kí hiệu tiệm cận Θ , O , và Ω

Định lý: Đối với hai hàm bất kỳ $f(n)$ và $g(n)$, ta có $f(n) = \Theta(g(n))$ khi và chỉ khi

$$f(n) = O(g(n)) \text{ và } f(n) = \Omega(g(n))$$

Định lý: Đối với hai hàm bất kỳ $f(n)$ và $g(n)$, ta có $f(n) = \Theta(g(n))$ khi và chỉ khi $f(n) = O(g(n))$ và $f(n) = \Omega(g(n))$

Ví dụ 1: Chứng minh $f(n) = 5n^2 = \Theta(n^2)$

Vì:

- $5n^2 = O(n^2)$

$f(n) = O(g(n))$ nếu tồn tại hằng số $c > 0$ và hằng số nguyên $n_0 \geq 1$ sao cho $f(n) \leq cg(n)$ với $n \geq n_0$

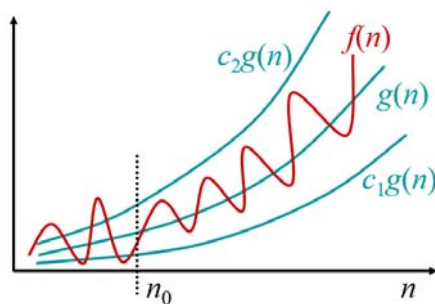
chọn $c = 5$ và $n_0 = 1$

- $5n^2 = \Omega(n^2)$

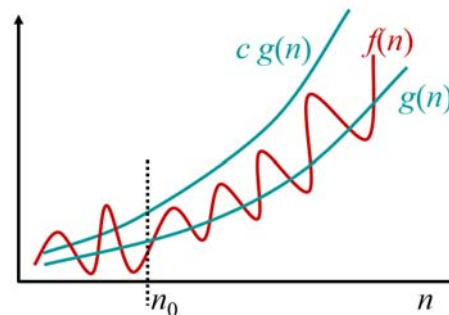
$f(n) = \Omega(g(n))$ nếu tồn tại hằng số $c > 0$ và hằng số nguyên $n_0 \geq 1$ sao cho $f(n) \geq cg(n)$ với $n \geq n_0$

Chọn $c = 5$ và $n_0 = 1$

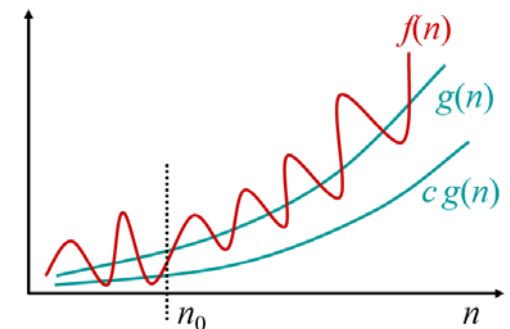
Do đó: $f(n) = \Theta(n^2)$



$$f(n) = \Theta(g(n))$$



$$f(n) = O(g(n))$$



$$f(n) = \Omega(g(n))$$

Định lý: Đối với hai hàm bất kỳ $f(n)$ và $g(n)$, ta có $f(n) = \Theta(g(n))$ khi và chỉ khi $f(n) = O(g(n))$ và $f(n) = \Omega(g(n))$

Ví dụ 2: Chứng minh $f(n) = 3n^2 - 2n + 5 = \Theta(n^2)$

Vì:

$$3n^2 - 2n + 5 = O(n^2)$$

$f(n) = O(g(n))$ nếu tồn tại hằng số $c > 0$ và số nguyên $n_0 \geq 1$ sao cho $f(n) \leq cg(n)$ với $n \geq n_0$

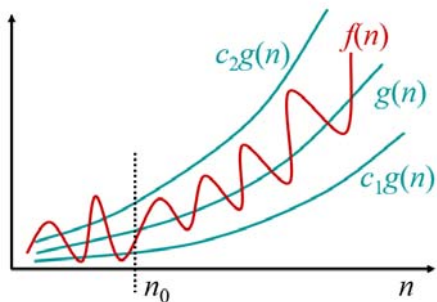
→ Chọn $c = ?$ và $n_0 = ?$

$$3n^2 - 2n + 5 = \Omega(n^2)$$

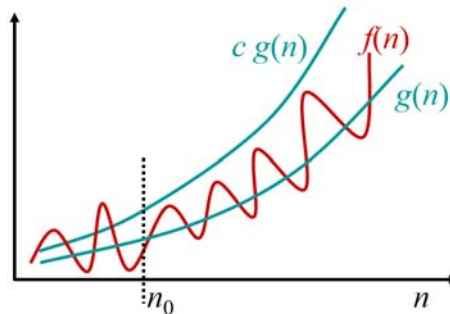
$f(n) = \Omega(g(n))$ nếu tồn tại hằng số $c > 0$ và số nguyên $n_0 \geq 1$ sao cho $f(n) \geq cg(n)$ với $n \geq n_0$

→ Chọn $c = ?$ và $n_0 = ?$

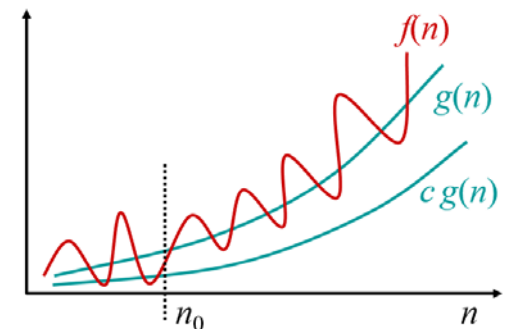
Do đó: $f(n) = \Theta(n^2)$



$$f(n) = \Theta(g(n))$$



$$f(n) = O(g(n))$$



$$f(n) = \Omega(g(n))$$

Bài tập 1

Chứng minh: $100n + 5 \neq \Omega(n^2)$

Giải: Chứng minh bằng phản chứng

– Giả sử: $100n + 5 = \Omega(n^2)$

→ $\exists c, n_0$ sao cho: $0 \leq cn^2 \leq 100n + 5$

– Ta có: $100n + 5 \leq 100n + 5n = 105n \quad \forall n \geq 1$

– Do đó: $cn^2 \leq 105n \Rightarrow n(cn - 105) \leq 0$

– Vì $n > 0 \Rightarrow cn - 105 \leq 0 \Rightarrow n \leq 105/c$

bất đẳng thức trên không đúng vì c phải là hằng số

Bài tập 2

Chứng minh: $n \neq \Theta(n^2)$

Giải: Chứng minh bằng phản chứng

– Giả sử: $n = \Theta(n^2)$

→ $\exists c_1, c_2, n_0$ sao cho: $c_1 n^2 \leq n \leq c_2 n^2 \quad \forall n \geq n_0$

→ $n \leq 1/c_1$

Bất đẳng thức trên không đúng vì c_1 phải là hằng số

Bài tập 3: Chứng minh

a) $6n^3 \neq \Theta(n^2)$

Giải: Chứng minh bằng phản chứng

– Giả sử: $6n^3 = \Theta(n^2)$

→ $\exists c_1, c_2, n_0$ sao cho: $c_1 n^2 \leq 6n^3 \leq c_2 n^2 \quad \forall n \geq n_0$

→ $n \leq c_2/6 \quad \forall n \geq n_0$

Bất đẳng thức trên không đúng vì c_2 phải là hằng số

b) $n \neq \Theta(\log_2 n)$

Giải:

Cách nói về thời gian tính

- Nói “Thời gian tính là $O(f(n))$ ”, hiểu là **đánh giá trong tình huống tồi nhất (worst case) là $O(f(n))$** (nghĩa là, không tồi hơn $c*f(n)$ với n lớn, vì kí hiệu O lớn cho ta cận trên). [Thường nói: “Đánh giá thời gian tính trong tình huống tồi nhất là $O(f(n))$ ”]
 - Nghĩa là thời gian tính trong tình huống tồi nhất được xác định bởi một hàm nào đó $g(n) \in O(f(n))$
- Nói “Thời gian tính là $\Omega(f(n))$ ”, hiểu là **đánh giá trong tình huống tốt nhất (best case) là $\Omega(f(n))$** (nghĩa là, không tốt hơn $c*f(n)$ với n lớn, vì kí hiệu Omega cho ta cận dưới). [Thường nói: “Đánh giá thời gian tính trong tình huống tốt nhất là $\Omega(f(n))$ ”]
 - Nghĩa là thời gian tính trong tình huống tốt nhất được xác định bởi một hàm nào đó $g(n) \in \Omega(f(n))$

Thời gian tính trong tình huống tồi nhất

- Khi phân tích một thuật toán
 - Chỉ quan tâm đến các trường hợp thuật toán chạy
 - Tồn thời gian chạy nhất
(tính cận trên cho thời gian tính của thuật toán)
 - Nếu thuật toán có thể giải trong thời gian cỡ hàm $f(n)$
 - Thì trường hợp tồi nhất, thời gian chạy không thể lớn hơn $c*f(n)$
- Hữu ích khi thuật toán áp dụng cho trường hợp
 - Cần biết cận trên cho thuật toán
- Ví dụ:
 - Các thuật toán áp dụng chạy trong nhà máy điện hạt nhân.

Thời gian tính trong tình huống tốt nhất

- Khi ta phân tích một thuật toán
 - Chỉ quan tâm đến các trường hợp thuật toán chạy
 - Ít thời gian nhất
(tính cận dưới của thời gian chạy)

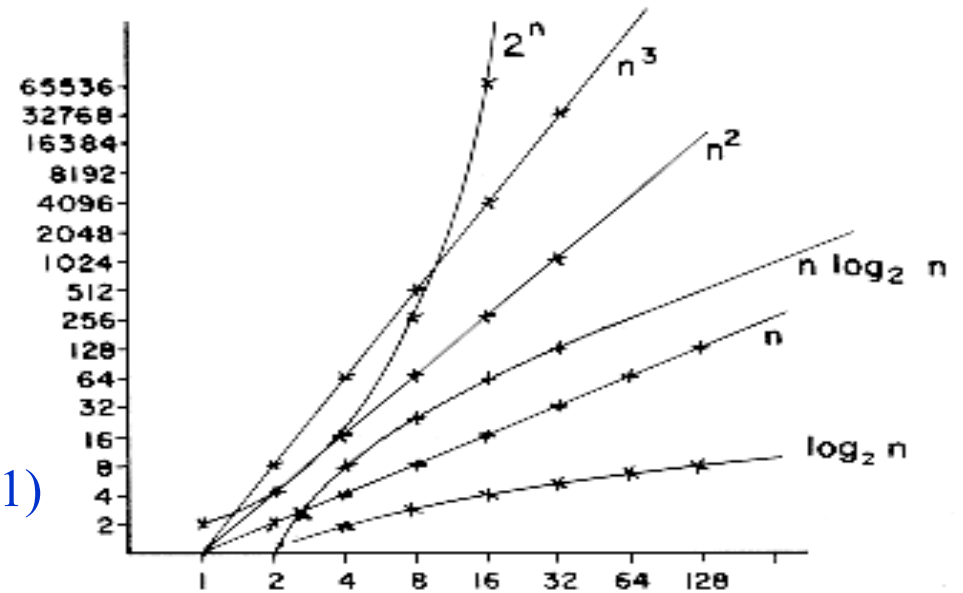
Thời gian tính trong tình huống trung bình

- Nếu thuật toán phải sử dụng nhiều lần
 - hữu ích khi tính được thời gian chạy trung bình của thuật toán với kích thước dữ liệu đầu vào là n
- Đánh giá thời gian tính trung bình khó hơn với việc đánh giá thời gian tính tồi/tốt nhất
 - Cần có thông tin về phân phối của dữ liệu dữ liệu

Ví dụ: Thuật toán sắp xếp chèn (Insertion sorting) có thời gian trung bình cỡ n^2

Một số lớp thuật toán cơ bản

- Hằng số $\approx O(1)$
- Logarithmic $\approx O(\log_2 n)$
- Tuyến tính $\approx O(n)$
- N-Log-N $\approx O(n \log_2 n)$
- Bình phương (Quadratic) $\approx O(n^2)$
- Bậc 3 (Cubic) $\approx O(n^3)$
- Hàm mũ (exponential) $\approx O(a^n)$ ($a > 1$)
- Đa thức (polynomial): $O(n^k)$ ($k \geq 1$)



- Thuật toán có đánh giá thời gian tính là $O(n^k)$ được gọi là **thuật toán thời gian tính đa thức** (hay vắn tắt: **thuật toán đa thức**)
 - Thuật toán đa thức được coi là thuật toán hiệu quả.
 - Các thuật toán với thời gian tính hàm mũ là không hiệu quả.
- Sau đây, ta sẽ lấy một số ví dụ về phân loại các hàm

Một số hàm cơ bản

Những hàm nào trong số những hàm sau giống nhau hơn?

$$n^{1000}$$

$$n^2$$

$$2^n$$

Đa thức
(polynomial)



Một số hàm cơ bản

Những hàm nào trong số những hàm sau giống nhau hơn?

$$1000n^2$$

$$3n^2$$

$$2n^3$$

Hàm bậc 2
(quadratic)



Tốc độ tăng của một số hàm cơ bản

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
4	2	4	8	16	64	16
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,094	262,144	$1.84 * 10^{19}$
128	7	128	896	16,384	2,097,152	$3.40 * 10^{38}$
256	8	256	2,048	65,536	16,777,216	$1.15 * 10^{77}$
512	9	512	4,608	262,144	134,217,728	$1.34 * 10^{154}$
1024	10	1,024	10,240	1,048,576	1,073,741,824	$1.79 * 10^{308}$

Phân loại thời gian tính của thuật toán

- Thời gian để giải một bộ dữ liệu đầu vào của
 - Thuật toán có thời gian tính tuyến tính (Linear Algorithm):
 - Không bao giờ lớn hơn $c*n$
 - Thuật toán có thời gian tính là hàm đa thức bậc 2 (Quadratic Algorithm):
 - Không bao giờ lớn hơn $c*n^2$
 - Thuật toán có thời gian tính là hàm đa thức bậc ba (Cubic Algorithm):
 - Không bao giờ lớn hơn $c*n^3$
 - Thuật toán có thời gian tính đa thức (Polynomial Algorithm)
 - Không bao giờ lớn hơn n^k
 - Thuật toán có thời gian tính hàm mũ (Exponential Algorithm)
 - Không bao giờ lớn hơn c^n
- với c và k là các hằng số tương ứng

Cận trên và cận dưới

Upper Bound and Lower Bound

- **Định nghĩa** (*Upper Bound*). Cho bài toán P , ta nói cận trên cho thời gian tính của P là $O(g(n))$ nếu để giải P tồn tại thuật toán giải với thời gian tính là $O(g(n))$.
- **Định nghĩa** (*Lower Bound*). Cho bài toán P , ta nói cận dưới cho thời gian tính của P là $\Omega(g(n))$ nếu mọi thuật toán giải P đều có thời gian tính là $\Omega(g(n))$.
- **Định nghĩa**. Cho bài toán P , ta nói thời gian tính của P là $\Theta(g(n))$ nếu P có cận trên là $O(g(n))$ và cận dưới là $\Omega(g(n))$.

Bài toán dễ giải, khó giải và không giải được

- Một bài toán được gọi là **dễ giải** (tractable) nếu như nó có thể giải được nhờ thuật toán đa thức. Bài toán có cận trên cho thời gian tính là đa thức.

Ví dụ: bài toán dãy con lớn nhất, bài toán sắp xếp dãy n số,

- Một bài toán được gọi là **khó giải** (intractable) nếu như nó không thể giải được bởi thuật toán đa thức. Bài toán có cận dưới cho thời gian tính là hàm mũ.

Ví dụ: Bài toán liệt kê các hoán vị của n số, các xâu nhị phân độ dài n , ...

- Một dạng bài toán nữa cũng được **coi là khó giải**: Đó là những bài toán cho đến thời điểm hiện tại vẫn chưa tìm được thuật toán đa thức để giải nó.

Ví dụ: Bài toán cái túi, Bài toán người du lịch,...

- Một bài toán được gọi là **không giải được** (unsolvable) nếu như không tồn tại thuật toán để giải nó..

Ví dụ: Bài toán về tính dừng, Bài toán về nghiệm nguyên của đa thức

Sự tương tự giữa so sánh các hàm số và so sánh số

Chú ý rằng mối quan hệ giữa các hàm số cũng giống như mối quan hệ “<, >, =” giữa các số

$$f \leftrightarrow g \approx a \leftrightarrow b$$

$$f(n) = O(g(n)) \approx a \leq b$$

$$f(n) = \Omega(g(n)) \approx a \geq b$$

$$f(n) = \Theta(g(n)) \approx a = b$$

$$f(n) = o(g(n)) \approx a < b$$

$$f(n) = \omega(g(n)) \approx a > b$$

Nhắc lại một số khái niệm toán học

■ Hàm mũ:

$$a^{(b+c)} = a^b a^c$$

$$a^{bc} = (a^b)^c$$

$$a^b / a^c = a^{(b-c)}$$

$$b = a^{\log_a b}$$

$$b^c = a^{c \cdot \log_a b}$$

■ Hàm Logarit:

$x = \log_b a$ là số mũ
cho $a = b^x$.

Logarit tự nhiên: $\ln a = \log_e a$

Logarit bậc 2: $\lg a = \log_2 a$

$$\lg^2 a = (\lg a)^2$$

$$\lg \lg a = \lg (\lg a)$$

$$a = b^{\log_b a}$$

$$\log_c (ab) = \log_c a + \log_c b$$

$$\log_b a^n = n \log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_b (1/a) = -\log_b a$$

$$\log_b a = \frac{1}{\log_a b}$$

$$a^{\log_b c} = c^{\log_b a}$$

Hàm logarit và hàm mũ

- Nếu cơ số của hàm logarit thay đổi giá trị từ hằng số này sang hằng số khác, thì giá trị của hàm bị thay đổi một lượng hằng số.
 - Ví dụ: $\log_{10} n * \log_2 10 = \log_2 n$.
 - Do đó, trong ký hiệu tiệm cận cơ số của log là không quan trọng:
$$O(\lg n) = O(\ln n) = O(\log n)$$
- Giá trị của hai hàm mũ khác nhau cơ số khác nhau một lượng cỡ hàm mũ (chứ không phải một lượng là hằng số)
 - Ví dụ: $2^n = (2/3)^n * 3^n$.

Bài tập

- Sắp xếp các hàm sau theo thứ tự tốc độ tăng dần

1. $n \log_2 n$

2. $\log_2 n^3$

3. n^2

4. $n^{2/5}$

5. $2^{\log_2 n}$

6. $\log_2(\log_2 n)$

7. $\text{Sqr}(\log_2 n)$

Sqr: square (bình phương)

Sqrt: square root (căn bậc 2)

Cách nhớ các kí hiệu

Theta	$f(n) = \Theta(g(n))$	$f(n) \approx c g(n)$
Big Oh	$f(n) = O(g(n))$	$f(n) \leq c g(n)$
Big Omega	$f(n) = \Omega(g(n))$	$f(n) \geq c g(n)$

Các tính chất

- Transitivity (truyền ứng)

$$f(n) = \Theta(g(n)) \ \& \ g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$$

$$f(n) = O(g(n)) \ \& \ g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$

$$f(n) = \Omega(g(n)) \ \& \ g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$$

- Reflexivity (phản xạ)

$$f(n) = \Theta(f(n)) \qquad f(n) = O(g(n)) \qquad f(n) = \Omega(g(n))$$

- Symmetry (đối xứng)

$$f(n) = \Theta(g(n)) \text{ khi và chỉ khi } g(n) = \Theta(f(n))$$

- Transpose Symmetry (Đối xứng chuyển vị)

$$f(n) = O(g(n)) \text{ khi và chỉ khi } g(n) = \Omega(f(n))$$

Ví dụ: $A = 5n^2 + 100n$, $B = 3n^2 + 2$. Chứng minh $A \in \Theta(B)$

Giải: $A \in \Theta(n^2)$, $n^2 \in \Theta(B) \Rightarrow A \in \Theta(B)$

Liên hệ với khái niệm giới hạn

- $\lim_{n \rightarrow \infty} [f(n) / g(n)] < \infty \Rightarrow f(n) \in O(g(n))$
- $0 < \lim_{n \rightarrow \infty} [f(n) / g(n)] < \infty \Rightarrow f(n) \in \Theta(g(n))$
- $0 < \lim_{n \rightarrow \infty} [f(n) / g(n)] \Rightarrow f(n) \in \Omega(g(n))$
- $\lim_{n \rightarrow \infty} [f(n) / g(n)]$ không xác định \Rightarrow không thể nói gì

Chú ý: $f(n) = n \sin n$; $g(n) = n$. Mặc dù $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \sin n$ không tồn tại, nhưng rõ ràng $n \sin n = O(n)$.

Ví dụ: Biểu diễn hàm A bằng kí hiệu tiệm cận sử dụng hàm B.

A

B

$$\log_3(n^2)$$

$$\log_2(n^3) \quad A \in \Theta(B)$$

$$\log_b a = \log_c a / \log_c b; A = 2 \lg n / \lg 3, B = 3 \lg n, A/B = 2/(3 \lg 3) \Rightarrow A \in \Theta(B)$$

Bài tập

Chứng minh

1) $3n^2 - 100n + 6 = O(n^2)$

2) $3n^2 - 100n + 6 = O(n^3)$

3) $3n^2 - 100n + 6 \neq O(n)$

4) $3n^2 - 100n + 6 = \Omega(n^2)$

5) $3n^2 - 100n + 6 \neq \Omega(n^3)$

6) $3n^2 - 100n + 6 = \Omega(n)$

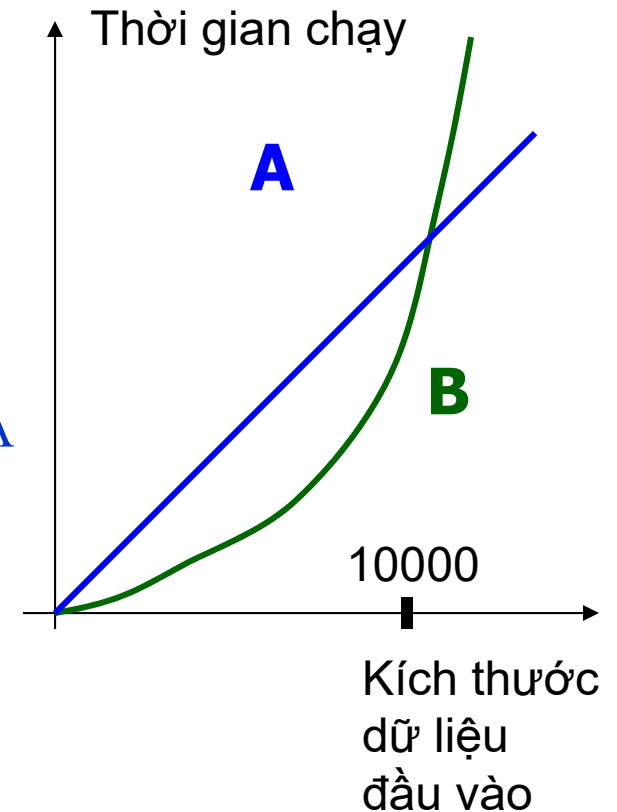
7) $3n^2 - 100n + 6 = \Theta(n^2)$

8) $3n^2 - 100n + 6 \neq \Theta(n^3)$

9) $3n^2 - 100n + 6 \neq \Theta(n)$

Chú ý

- Giả sử có 2 thuật toán:
 - Thuật toán A có thời gian chạy $30000n$
 - Thuật toán B có thời gian chạy $3n^2$
- Theo đánh giá tiệm cận, thuật toán A tốt hơn thuật toán B
- Tuy nhiên, nếu kích thước dữ liệu của bài toán luôn nhỏ hơn 10000, thì thuật toán B lại tốt (nhanh) hơn A



Nội dung

1.1. Ví dụ mở đầu

1.2. Thuật toán và độ phức tạp

1.3. Kí hiệu tiệm cận

1.4. Giả ngôn ngữ (Pseudo code)

1.5. Một số kĩ thuật phân tích thuật toán

1.6. Giải công thức đệ quy

1.4. Giả ngôn ngữ (Pseudocode)

- Để mô tả thuật toán, ta có thể sử dụng một ngôn ngữ lập trình nào đó. Tuy nhiên, điều đó có thể làm cho việc mô tả thuật toán trở nên phức tạp và khó nắm bắt. Do đó, để mô tả thuật toán, người ta thường sử dụng giả ngôn ngữ (pseudo language), cho phép:
 - Mô tả thuật toán bằng ngôn ngữ đời thường
 - Sử dụng các cấu trúc câu lệnh tương tự như của ngôn ngữ lập trình.
- Dưới đây ta liệt kê một số câu lệnh chính được sử dụng trong giáo trình để mô tả thuật toán.

Ví dụ: tìm phần tử lớn nhất trong mảng

```
Function arrayMax(A, n)  
//Input: mảng A gồm n số nguyên  
//Output: phần tử lớn nhất của mảng A  
begin  
    currentMax  $\leftarrow$  A[0]  
    for i  $\leftarrow$  1 to n - 1  
        if (A[i] > currentMax) then  
            currentMax  $\leftarrow$  A[i]  
        endif;  
    endfor;  
    return currentMax;  
end;
```


1.4. Pseudocode

- **Khai báo biến**

integer x, y;

real u, v;

boolean a, b;

char c, d;

datatype x;

- **Lệnh gán**

x = biểu thức;

hoặc

$x \leftarrow$ biểu thức;

Ví dụ: $x \leftarrow 1+4$; $y = a*y+2$;

1.4. Pseudocode

Cấu trúc điều khiển:

if điều_kiện **then**

dãy các câu lệnh

else

dãy các câu lệnh

endif;

while điều_kiện **do**

dãy các câu lệnh

endwhile;

repeat

dãy các câu lệnh

until condition;

1.4. Pseudocode

Cấu trúc điều khiển:

for i=n1 **to** n2 [step d]

dãy các câu lệnh

endfor;

Case

điều_kiện_1: câu_lệnh_1;

điều_kiện_2: câu_lệnh_2;

.

.

.

điều_kiện_n: câu_lệnh_n;

endcase;

1.4. Pseudocode

- **Vào/ra:**

read(X); /* X là biến */

print(data);

hoặc **print**(thông báo);

- **Hàm và thủ tục:**

Function name(arguments)

begin

 khai báo biến;

 các câu lệnh trong hàm;

return (value);

end;

Procedure name(arguments)

begin

 khai báo biến;

 các câu lệnh trong thủ tục;

end;

Ví dụ: tìm phần tử lớn nhất trong mảng

Function *arrayMax*(*A*, *n*)

//Input: mảng *A* gồm *n* số nguyên

//Output: phần tử lớn nhất của mảng *A*

begin

currentMax \leftarrow *A*[0]

for *i* \leftarrow 1 **to** *n* - 1

if (*A*[*i*] > *currentMax*) **then**

currentMax \leftarrow *A*[*i*]

endif;

endfor;

return *currentMax*;

end;

1.4. Pseudocode

Ví dụ 2: hoán đổi giá trị của hai biến

```
Procedure swap(x, y)
begin
    temp=x;
    x = y;
    y = temp;
end;
```

Hoặc có thể viết đơn giản như sau:

```
Procedure swap(x, y)
    temp=x;
    x = y;
    y = temp;
```

Pseudocode: Ví dụ 3

Đề bài: Tìm số nguyên tố nhỏ nhất lớn hơn một số nguyên dương n cho trước

- Đầu tiên ta viết hàm **Is_Prime** kiểm tra xem một số nguyên dương m có phải là số nguyên tố hay không. Xây dựng hàm này như sau:
 - Nếu $m = a*b$ với $1 < a, b < m$, thì một trong hai số a và b có giá trị không thể vượt quá \sqrt{m} . Do đó, số nguyên tố lớn nhất nhỏ hơn m sẽ có giá trị không vượt quá $\sqrt{m} \rightarrow m$ là số nguyên tố nếu nó không chia hết cho bất kì số nguyên nào trong khoảng $[2, \sqrt{m}]$.
- Sau đó, để tìm số nguyên tố nhỏ nhất lớn hơn n như sau:
 - Ta dùng hàm **Is_Prime** để xét lần lượt các số lớn hơn n là: $n+1, n+2, n+3, \dots$ xem có số nào là số nguyên tố hay không. Nếu tìm được thì thuật toán kết thúc và trả về số nguyên tố vừa tìm được.

Pseudocode: Ví dụ 3

Thuật toán kiểm tra số nguyên dương có phải là số nguyên tố hay không:

- **Input:** Số nguyên dương m .
- **Output:** **true** nếu m là số nguyên tố, **false** nếu ngược lại.

```
Function Is_prime( $m$ )  
begin  
     $i = 2$ ;  
    while ( $i*i \leq m$ ) and ( $m \bmod i \neq 0$ ) do  
         $i=i+1$ ;  
    endwhile;  
    Is_prime =  $i > \text{sqrt}(m)$ ;  
end Is_Prime;
```

Thuật toán tìm số nguyên tố nhỏ nhất mà lớn hơn số nguyên dương n :

- Thuật toán sử dụng hàm Is_prime như là 1 thủ tục con.
- **Input:** Số nguyên dương n .
- **Output:** m là số nguyên tố nhỏ nhất mà lớn hơn số nguyên dương n .

```
procedure Lagre_Prime( $n$ )  
begin  
     $m = n+1$ ;  
    while not Is_prime( $m$ ) do  
         $m=m+1$ ;  
    endwhile;  
end;
```

Nội dung

1.1. Ví dụ mở đầu

1.2. Thuật toán và độ phức tạp

1.3. Kí hiệu tiệm cận

1.4. Giả ngôn ngữ (Pseudo code)

1.5. Một số kĩ thuật phân tích thuật toán

1.6. Giải công thức đệ quy

Tính toán thời gian tính của thuật toán

- **Đánh giá qua thời gian chạy thực nghiệm:**

- Cần: cài đặt chương trình thực thi thuật toán, sau đó chạy chương trình và đo thời gian chạy.
- Nhược điểm:
 - Bắt buộc phải cài đặt thuật toán
 - Kết quả thu được sẽ không bao gồm thời gian chạy của những dữ liệu đầu vào không được chạy thực nghiệm. Do vậy, cần chọn được các bộ dữ liệu thử đặc trưng cho tất cả tập các dữ liệu vào của thuật toán → rất khó khăn và tốn nhiều chi phí.
 - Để so sánh thời gian tính của hai thuật toán, cần phải chạy thực nghiệm hai thuật toán trên cùng một máy, và sử dụng cùng một phần mềm.

→ Ta cần đánh giá thuật toán theo hướng xấp xỉ tiệm cận

- **Đánh giá thời gian chạy theo hướng xấp xỉ tiệm cận:**

- Sử dụng giả ngôn ngữ để mô tả thuật toán, thay vì cài đặt thực sự
- Phân tích thời gian tính như làm hàm của dữ liệu đầu vào, n
- Đánh giá trên tất cả các bộ dữ liệu vào có thể có của bài toán
- Cho phép ta đánh giá được thời gian tính của thuật toán độc lập với phần cứng và phần mềm cần sử dụng để cài đặt thuật toán.

Phép toán cơ bản

- Để đo thời gian tính bằng phương pháp đánh giá tiệm cận, ta sẽ đếm số **phép toán cơ bản** mà thuật toán phải thực hiện

là phép toán có thể thực hiện với thời gian bị chặn bởi một hằng số không phụ thuộc vào kích thước dữ liệu vào.

- Ví dụ về các phép toán cơ bản:

- Tính biểu thức
- Phép gán giá trị cho một biến
- Đánh chỉ số mảng
- Gọi 1 phương thức
- Lệnh trả về giá trị từ 1 phương thức

$x^2 + e^y$

$cnt \leftarrow cnt + 1$

$A[5]$

$mySort(A, n)$

$return(cnt)$

Phân tích độ phức tạp của thuật toán: Các kĩ thuật cơ bản

1. Cấu trúc tuần tự

- Thời gian tính của chương trình “**P; Q**”, với P và Q là hai đoạn chương trình thực thi một thuật toán, P thực hiện trước, rồi đến Q là

$$Time(P; Q) = Time(P) + Time(Q) ,$$

hoặc ta có thể dùng kí hiệu tiệm cận theta:

$$Time(P; Q) = \Theta(\max(Time(P), Time(Q))).$$

với $Time(P)$, $Time(Q)$ là thời gian tính của P và Q.

2. Vòng lặp FOR

for $i = 1$ **to** m **do** $P(i)$;

Giả sử thời gian thực hiện $P(i)$ là $t(i)$, khi đó thời gian thực hiện vòng lặp for là $\sum_{i=1}^m t(i)$

3. Vòng lặp lồng nhau

for $i = 1$ **to** n **do**

for $j = 1$ **to** m **do** $P(j)$;

Giả sử thời gian thực hiện $P(j)$ là $t(j)$, khi đó thời gian thực hiện vòng lặp lồng nhau này là:

Phân tích độ phức tạp của thuật toán: Các kĩ thuật cơ bản

4. If/Else

if (điều_kiện) then P;

else Q;

endif;

Thời gian thực hiện câu lệnh if/else

= thời gian kiểm tra (điều_kiện) + $\max(\text{Time}(P), \text{Time}(Q))$

Ví dụ

Case1: for (i=0; i<n; i++)
 for (j=0; j<n; j++)
 k++;

$O(n^2)$

$O(n^2)$

$O(n^2)$

Phân tích độ phức tạp của thuật toán: Các kĩ thuật cơ bản

5. Vòng lặp while/repeat

- Cần xác định một hàm của các biến trong vòng lặp sao cho hàm này có giá trị giảm dần trong quá trình lặp. Khi đó:
 - Để chứng minh tính kết thúc của vòng lặp ta chỉ cần chỉ ra giá trị của hàm là số nguyên dương.
 - Còn để xác định số lần lặp ta cần phải khảo sát xem giá trị của hàm giảm như thế nào.
- Việc phân tích vòng lặp Repeat được tiến hành tương tự như phân tích vòng lặp While.

Câu lệnh đặc trưng

- **Định nghĩa.** *Câu lệnh đặc trưng là câu lệnh được thực hiện thường xuyên ít nhất là cũng như bất kỳ câu lệnh nào trong thuật toán.*
- Nếu giả thiết thời gian thực hiện mỗi câu lệnh là bị chặn bởi hằng số thì thời gian tính của thuật toán sẽ cùng cỡ với số lần thực hiện câu lệnh đặc trưng
- \Rightarrow Để đánh giá thời gian tính có thể đếm số lần thực hiện câu lệnh đặc trưng

Ví dụ 1: Tính dãy Fibonacci

```
function Fibrec(n)
  if  $n < 2$  then return  $n$ ;
  else return  $\text{Fibrec}(n-1) + \text{Fibrec}(n-2)$ ;
```

```
function Fibiter(n)
   $i=0$ ;
   $j=1$ ;
  for  $k=1$  to  $n$  do
     $j=i+j$ ;
     $i=j-i$ ;
  return  $j$ ;
```

Câu lệnh đặc trưng

- Số lần thực hiện câu lệnh đặc trưng là $n \rightarrow$ Thời gian chạy Fibiter là $O(n)$

- Dãy Fibonacci (0, 1, 1, 2, 3, 5, 8, 13, 21, 34....)
 - $f_0=0$;
 - $f_1=1$;
 - $f_n = f_{n-1} + f_{n-2}$

n	10	20	30	50	100
Fibrec	8ms	1sec	2min	21days	10^9 years
Fibiter	0.17ms	0.33ms	0.5ms	0.75ms	1.5ms

Ví dụ 2: Bài toán dãy con lớn nhất

Cho mảng số nguyên A_1, A_2, \dots, A_N , tìm giá trị lớn nhất của $\sum_{k=i}^j A_k$

Thuật toán 1. Duyệt toàn bộ (Brute force)

```
int maxSum = 0;
for (int i=0; i<n; i++) {
    for (int j=i; j<n; j++) {
        int sum = 0;
        for (int k=i; k<=j; k++)
            sum += a[k];
        if (sum > maxSum)
            maxSum = sum;
    }
}
```

Chọn câu lệnh đặc trưng là **sum+=a[k]**

➔ Thời gian tính của thuật toán: $O(n^3)$

Thuật toán 2. Duyệt toàn bộ có cải tiến

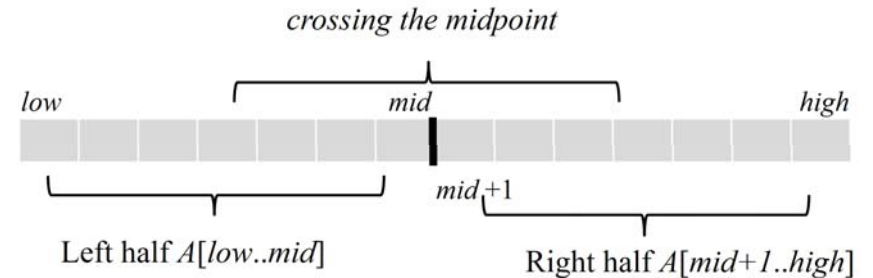
```
int maxSum = a[0];
for (int i=0; i<n; i++) {
    int sum = 0;
    for (int j=i; j<n; j++) {
        sum += a[j];
        if (sum > maxSum)
            maxSum = sum;
    }
}
```

$O(n^2)$

Thuật toán 3. Đệ quy

```
MaxLeft(a, low, mid);
{
    maxSum = -∞; sum = 0;
    for (int k=mid; k>=low; k--) {
        sum = sum+a[k];
        maxSum = max(sum, maxSum);
    }
    return maxSum;
}
```

← $O(n)$



```
MaxRight(a, mid, high);
{
    maxSum = -∞; sum = 0;
    for (int k=mid; k<=high; k++){
        sum = sum+a[k];
        maxSum = max(sum, maxSum);
    }
    return maxSum;
}
```

← $O(n)$

```
MaxSub(a, low, high);
{
    if (low = high) return a[low] //base case: only 1 element
    else
    {
        mid = (low+high)/2;
        wL = MaxSub(a, low, mid); ←  $T(n/2)$ 
        wR = MaxSub(a, mid+1, high); ←  $T(n/2)$ 
        wM = MaxLeft(a, low, mid) + MaxRight(a, mid, high);
        return max(wL, wR, wM);
    }
}
```

$$T(n) = 2T(n/2) + O(n)$$
$$\rightarrow T(n) = O(n \log n)$$

Giải công thức đệ quy này thế nào ? (xem mục 1.6.)

Nhắc lại một số kiến thức

$$S(N) = 1 + 2 + \dots + N = \sum_{i=1}^N i = N(1 + N) / 2$$

- Tổng bình phương: $\sum_{i=1}^N i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3}$ for large N

- Tổng mũ: $\sum_{i=1}^N i^k \approx \frac{N^{k+1}}{|k+1|}$ for large N and $k \neq -1$

- Dãy: $\sum_{i=0}^N A^i = \frac{A^{N+1} - 1}{A - 1}$

- Trường hợp đặc biệt $A = 2$

- $2^0 + 2^1 + 2^2 + \dots + 2^N = 2^{N+1} - 1$

Ví dụ 3

- Đưa ra đánh giá tiệm cận O lớn cho thời gian tính $T(n)$ của đoạn chương trình sau:

```
for (int i = 1; i<=n; i++)  
    for (int j = 1; j<= i*i*i; j++)  
        for (int k = 1; k<=n; k++)  
            x = x + 1;
```

- Giải:

$$\begin{aligned} T(n) &= \sum_{i=1}^n \sum_{j=1}^{i^3} \sum_{k=1}^n 1 \\ &= \sum_{i=1}^n \sum_{j=1}^{i^3} n = \sum_{i=1}^n n \left(\sum_{j=1}^{i^3} 1 \right) = \sum_{i=1}^n n i^3 \\ &= n \sum_{i=1}^n i^3 \leq n \sum_{i=1}^n n^3 = n^4 \sum_{i=1}^n 1 = n^5 \end{aligned}$$

So $T(n) = O(n^5)$.

Ví dụ 4

Đưa ra đánh giá tiệm cận O lớn cho thời gian tính $T(n)$ của đoạn chương trình sau:

```
a)   int x = 0;
      for (int i = 1; i <= n; i *= 2)
          x = x + 1;
```

- Giải:

Vòng lặp **for** thực hiện $\log_2 n$ lần, do đó $T(n) = O(\log_2 n)$.

```
b)   int x = 0;
      for (int i = n; i > 0; i /= 2)
          x = x + 1;
```

- Giải:

Vòng lặp **for** thực hiện

Ví dụ 5

Đưa ra đánh giá tiệm cận O lớn cho thời gian tính $T(n)$ của đoạn chương trình sau:

```
int n;  
if (n<1000)  
    for (int i=0; i<n; i++)  
        for (int j=0; j<n; j++)  
            for (int k=0; k<n; k++)  
                cout << "Hello\n";  
else  
    for (int j=0; j<n; j++)  
        for (int k=0; k<n; k++)  
            cout << "world!\n";
```

Giải:

- $T(n)$ hằng số khi $n < 1000$. $T(n) = O(n^2)$.

Ví dụ 6: Thuật toán số nguyên tố

- Thuật toán kiểm tra xem số nguyên dương $m > 1$ có phải là số nguyên tố hay không.

Algorithm PRIME(m)

for $i=2$ **to** $\text{sqrt}(m)$ **do**

if $m \bmod i = 0$ **then return FALSE**

return YES

- Thời gian tính: $T(n) = O(\sqrt{m})$. Thuật toán đa thức?
- Kích thước dữ liệu vào: $n \approx \log_2 m$

$$\Rightarrow m \approx 2^n$$

$$\Rightarrow T(n) = O(m^{1/2}) = O(2^{n/2}).$$

Thời gian tính hàm mũ!

Nội dung

1.1. Ví dụ mở đầu

1.2. Thuật toán và độ phức tạp

1.3. Kí hiệu tiệm cận

1.4. Giả ngôn ngữ (Pseudo code)

1.5. Một số kĩ thuật phân tích thuật toán

1.6. Giải công thức đệ quy

Độ phức tạp tính toán của các thuật toán được xây dựng dưới dạng công thức đệ quy của số lượng thao tác trong thuật toán.

Ví dụ: Trong mục trước, ta học thuật toán đệ quy giải bài toán tổng dãy con lớn nhất có độ phức tạp là $T(n) = 2T(n/2) + O(n)$.

Giải công thức đệ quy này ta thu được $T(n) = O(n \log n)$

1.6. Giải công thức đệ quy

Định nghĩa:

Công thức đệ quy cho dãy số $\{a_n\}$ là công thức biểu diễn a_n dưới dạng một hoặc nhiều thành phần trước nó trong dãy a_0, a_1, \dots, a_{n-1} với mọi số nguyên $n \geq n_0$, n_0 là số nguyên không âm

Một dãy số được gọi là một **lời giải** của công thức đệ quy nếu các thành phần trong dãy số đó thỏa mãn công thức đệ quy.

Ví dụ: Xét công thức đệ quy

$$a_n = 2a_{n-1} - a_{n-2} \text{ với } n = 2, 3, 4, \dots \quad \text{Dãy số } a_n = n+1??$$

- Dãy số $a_n = 3n$ là lời giải của công thức đệ quy đã cho?

Với $n \geq 2$ ta có: $2a_{n-1} - a_{n-2} = 2(3(n-1)) - 3(n-2) = 3n = a_n$

Do đó, dãy số $a_n = 3n$ là lời giải của công thức đệ quy đã cho

- Dãy số $a_n = 5$ cũng là lời giải của công thức đệ quy này?

Với $n \geq 2$ ta có: $2a_{n-1} - a_{n-2} = 2 \cdot 5 - 5 = 5 = a_n$

Do đó, dãy $a_n = 5$ cũng là lời giải của công thức đệ quy đã cho

Cùng 1 công thức đệ quy có thể có **nhiều lời giải**. ➔ Tại sao???

1.6. Giải công thức đệ quy

Một công thức đệ quy **không có các giá trị đầu (các điều kiện đầu)**.

→ Có thể có (thường có) **nhiều lời giải**

Ví dụ: $a_n = 2a_{n-1} - a_{n-2}$ với $n = 2, 3, 4, \dots$. Công thức đệ quy này có các lời giải:

- $a_n = 5$
- $a_n = 3n$
- $a_n = n + 1$

Nếu **cả** công thức đệ quy và các điều kiện đầu đều được xác định, thì dãy số lời giải của công thức đệ quy sẽ được xác định **duy nhất**.

Ví dụ: $a_n = 2a_{n-1} - a_{n-2}$ với $n = 2, 3, 4, \dots$
với $a_0 = 0; a_1 = 3$

→ Dãy số $a_n = 5$ không phải là lời giải

→ Dãy số $a_n = 3n$ là lời giải duy nhất

Công thức đệ quy là công thức cho phép tính giá trị của các đại lượng theo từng bước, dựa vào các giá trị tính ở các bước trước và một số giá trị đầu.

1.6. Giải công thức đệ quy

Ta hiểu việc giải công thức đệ quy là việc tìm công thức dưới dạng hiện cho số hạng tổng quát của dãy số thoả mãn công thức đệ quy đã cho.

Ví dụ: Cho công thức đệ quy:

$$a_n = 2a_{n-1} - a_{n-2} \text{ với } n = 2, 3, 4, \dots$$

$$a_0 = 0; a_1 = 3$$

→ Công thức dạng hiện của công thức đệ quy trên là dãy số $a_n = 3n$

→ $a_n = 3n$ được gọi là nghiệm (lời giải) của công thức đệ quy trên

- Chưa có phương pháp giải mọi công thức đệ quy.
- Xét một số phương pháp giải:
 - Phương trình đặc trưng giải Công thức đệ quy tuyến tính thuần nhất hệ số hằng (sẽ viết tắt là CTĐQ TTTNHS)
 - Phương pháp thế xuôi
 - Phương pháp thế ngược
 - Cây đệ quy

1.6. Giải công thức đệ quy: Phương pháp Phương trình đặc trưng

Định nghĩa. Công thức đệ quy tuyến tính thuần nhất hệ số hằng bậc k là công thức đệ quy sau

$$a_n = c_1 a_{n-1} + \dots + c_k a_{n-k}$$

trong đó c_i là các hằng số, và $c_k \neq 0$.

Giải thích:

- Tuyến tính: về phải là tổng của các số hạng trước số hạng an trong dãy số với các hệ số (c_1, c_2, \dots, c_k) là hằng số (không phải là hàm phụ thuộc vào n)
- Thuần nhất: về phải không có thêm số hạng nào khác với các số hạng a_j của dãy
- Bậc k : về phải có số hạng thứ $(n-k)$ của dãy

Dãy số thoả mãn công thức đã cho là **xác định duy nhất** nếu đòi hỏi nó thoả mãn k điều kiện đầu: $a_0 = C_0, a_1 = C_1, \dots, a_{k-1} = C_{k-1}$,

trong đó C_0, C_1, \dots, C_{k-1} là các hằng số.

Ví dụ 1: $a_n = 2a_{n-1} - a_{n-2}$ với $n = 2, 3, 4, \dots$ Công thức đệ quy này có một số lời giải như sau:

- $a_n = 5$ $a_n = 3n$ $a_n = n + 1$

Ví dụ 2: $a_n = 2a_{n-1} - a_{n-2}$ với $n = 2, 3, 4, \dots$ và các điều kiện đầu: $a_0 = 0; a_1 = 3$

- ➔ Dãy $a_n = 5$ không là lời giải của công thức đệ quy đã cho
- ➔ Dãy $a_n = 3n$ là lời giải của công thức đệ quy đã cho

Công thức đệ quy tuyến tính thuần nhất hệ số hằng

- **Ví dụ1:** Đây là CTĐQ TTTNHS

1) ~~$a_n = 4a_{n-1} + 2na_{n-3}$~~

2) ~~$h_n = 2h_{n-1} + 1$~~

3) ~~$b_n = 5b_{n-2} + 2(b_{n-3})^2$~~

4) $q_n = 3q_{n-6} + q_{n-8}$

Định nghĩa. Công thức đệ quy tuyến tính thuần nhất hệ số hằng bậc k là công thức đệ quy sau

$$a_n = c_1 a_{n-1} + \dots + c_k a_{n-k}$$

trong đó c_i là các hằng số, và $c_k \neq 0$.

CTĐQ TTTNHSK bậc k

Ví dụ 2:

- Công thức đệ quy $P_n = (1.05)P_{n-1}$
Là công thức đệ quy tuyến tính thuần nhất hệ số hằng **bậc 1**.
- Công thức đệ quy $f_n = f_{n-1} + f_{n-2}$
Là công thức đệ quy tuyến tính thuần nhất hệ số hằng **bậc 2**.
- Công thức đệ quy $a_n = a_{n-5}$
Là công thức đệ quy tuyến tính thuần nhất hệ số hằng **bậc 5**.

Giải CTĐQ TTTNHS

- Ta sẽ tìm nghiệm dưới dạng $a_n = r^n$, trong đó r là hằng số.
- Dãy số $\{a_n = r^n\}$ thoả mãn CTĐQ đã cho

$$a_n = c_1 a_{n-1} + \dots + c_k a_{n-k}$$

nếu r thoả mãn phương trình:

$$r^n = c_1 r^{n-1} + \dots + c_k r^{n-k}, \text{ hay} \quad (\text{chuyển về}$$

$r^k - c_1 r^{k-1} - \dots - c_k = 0$

$$\text{và } \times \text{ với } r^{k-n})$$

phương trình đặc trưng, còn nghiệm của nó sẽ được gọi là nghiệm đặc trưng của CTĐQ TTTNHS.

- Ta có thể sử dụng nghiệm đặc trưng để thu được công thức cho dãy số.

Giải CTĐQ TTTNHS

Xét công thức đệ quy tuyến tính thuần nhất hệ số hằng **bậc 2**:

$$a_n = c_1 a_{n-1} + c_2 a_{n-2}$$

Định lý 1. Cho c_1, c_2 là các hằng số thực.

Giả sử phương trình $r^2 - c_1 r - c_2 = 0$ có 2 nghiệm phân biệt r_1 và r_2 . Khi đó dãy số $\{a_n\}$ là nghiệm của công thức đệ quy

$$a_n = c_1 a_{n-1} + c_2 a_{n-2}$$

khi và chỉ khi

$$a_n = \alpha_1 (r_1)^n + \alpha_2 (r_2)^n \quad (1)$$

$n = 0, 1, \dots$, trong đó α_1, α_2 là các hằng số.

Giải CTĐQ TTTNHS

- **Chứng minh.** Trước hết ta chứng minh rằng nếu r_1 và r_2 là hai nghiệm phân biệt của phương trình đặc trưng, và α_1 , α_2 là các hằng số, thì dãy số $\{a_n\}$ xác định bởi công thức (1) là nghiệm của công thức đệ quy đã cho
- Thực vậy, do r_1 và r_2 là nghiệm đặc trưng nên

$$r_1^2 = c_1 r_1 + c_2 ,$$

$$r_2^2 = c_1 r_2 + c_2$$

Giải CTĐQ TTTNHS

- Từ đó suy ra

$$\begin{aligned}c_1 a_{n-1} + c_2 a_{n-2} &= c_1 (\alpha_1 r_1^{n-1} + \alpha_2 r_2^{n-1}) + c_2 (\alpha_1 r_1^{n-2} + \alpha_2 r_2^{n-2}) \\&= \alpha_1 r_1^{n-2}(c_1 r_1 + c_2) + \alpha_2 r_2^{n-2}(c_1 r_2 + c_2) \\&= \alpha_1 r_1^{n-2} r_1^2 + \alpha_2 r_2^{n-2} r_2^2 \\&= \alpha_1 r_1^n + \alpha_2 r_2^n \\&= a_n .\end{aligned}$$

Giải CTĐQ TTTNHS

- Bây giờ ta sẽ chỉ ra rằng nghiệm $\{a_n\}$ của công thức đệ quy

$a_n = c_1 a_{n-1} + c_2 a_{n-2}$ luôn có dạng (1) với α_1, α_2 nào đó.

- Thực vậy, giả sử $\{a_n\}$ là nghiệm của công thức đệ quy đã cho với điều kiện đầu

$$a_0 = C_0, a_1 = C_1, \quad (2)$$

- Ta chỉ ra rằng có thể tìm được các số α_1, α_2 để cho (1) là nghiệm của hệ thức với điều kiện đầu này

Giải CTĐQ TTTNHS

- Ta có

$$a_0 = C_0 = \alpha_1 + \alpha_2 ,$$

$$a_1 = C_1 = \alpha_1 r_1 + \alpha_2 r_2 .$$

- Hệ phương trình tuyến tính phụ thuộc hai ẩn α_1, α_2 này có định thức là $r_2 - r_1 \neq 0$ (do $r_1 \neq r_2$) có nghiệm duy nhất

$$\alpha_1 = (C_1 - C_0 r_2)/(r_1 - r_2), \quad \alpha_2 = (C_0 r_1 - C_1)/(r_1 - r_2).$$

- Với những giá trị của α_1, α_2 vừa tìm được, dãy $\{a_n\}$ xác định theo (1) là nghiệm của hệ thức đã cho với điều kiện đầu (2). Do hệ thức đã cho cùng với điều kiện đầu (2) xác định duy nhất một dãy số, nên nghiệm của hệ thức được cho bởi công thức (1).

Định lý được chứng minh

Giải CTĐQ TTTNHS

Ví dụ 1: Tìm lời giải cho công thức đệ quy

$$a_n = a_{n-1} + 2a_{n-2} \text{ với } a_0 = 2 \text{ và } a_1 = 7 ?$$

• **Giải:** Phương trình đặc trưng của công thức đệ quy là

$$r^2 - r - 2 = 0.$$

Có 2 nghiệm phân biệt $r_1 = 2$ và $r_2 = -1$.

Do đó, dãy $\{a_n\}$ là lời giải của công thức đệ quy khi và chỉ khi:

$$a_n = \alpha_1 2^n + \alpha_2 (-1)^n \text{ với giá trị nào đó của } \alpha_1 \text{ và } \alpha_2.$$

Cho biểu thức $a_n = \alpha_1 2^n + \alpha_2 (-1)^n$ và các điều kiện đầu $a_0 = 2$ và $a_1 = 7$, ta có

$$a_0 = 2 = \alpha_1 + \alpha_2$$

$$a_1 = 7 = \alpha_1 \cdot 2 + \alpha_2 \cdot (-1)$$

Giải hệ phương trình này ta có:

$$\alpha_1 = 3 \text{ và } \alpha_2 = -1.$$

Do đó, lời giải của công thức đệ quy với điều kiện đầu đã cho là dãy $\{a_n\}$ với

$$a_n = 3 \cdot 2^n - (-1)^n$$

Giải CTĐQ TTTNHS

Ví dụ 2: Tìm lời giải cho công thức đệ quy (dãy Fibonacci)

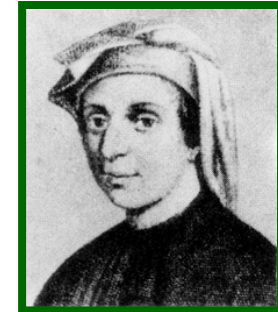
$$F_n = F_{n-1} + F_{n-2}, n \geq 2,$$

$$F_0 = 0, F_1 = 1$$

Giải: Phương trình đặc trưng của công thức đệ quy

$$r^2 - r - 1 = 0.$$

Có 2 nghiệm phân biệt là



Leonardo Fibonacci
1170-1250

Trường hợp nghiệm kép

Định lý 2: Cho c_1, c_2 là các hằng số thực, $c_2 \neq 0$. Giả sử phương trình $r^2 - c_1 r - c_2 = 0$ có nghiệm kép r_0 . Khi đó dãy số $\{a_n\}$ là nghiệm của công thức đệ qui $a_n = c_1 a_{n-1} + c_2 a_{n-2}$

khi và chỉ khi

$$a_n = \alpha_1 r_0^n + \alpha_2 n r_0^n$$

$n = 0, 1, \dots$, trong đó α_1, α_2 là các hằng số.

Ví dụ 3

Tìm nghiệm cho công thức đệ quy

$$a_n = 6 a_{n-1} - 9 a_{n-2}$$

với điều kiện đầu $a_0 = 1$ và $a_1 = 6$.

Giải:

Phương trình đặc trưng:

$r^2 - 6r + 9 = 0$ có nghiệm kép $r = 3$. Do đó nghiệm của hệ thức có dạng:

$$a_n = \alpha_1 3^n + \alpha_2 n 3^n.$$

Để tìm α_1, α_2 , sử dụng điều kiện đầu ta có

$$a_0 = 1 = \alpha_1,$$

$$a_1 = 6 = \alpha_1 \cdot 3 + \alpha_2 \cdot 1 \cdot 3$$

Giải hệ này ta tìm được $\alpha_1 = 1$ và $\alpha_2 = 1$.

Từ đó nghiệm của hệ thức đã cho là:

$$a_n = 3^n + n 3^n.$$

$$\text{CTĐQ: } a_n = c_1 a_{n-1} + \dots + c_k a_{n-k}$$

$$\text{Phương trình đặc trưng: } r^k - c_1 r^{k-1} - \dots - c_k = 0$$

Trường hợp tổng quát

Định lý 3. Cho c_1, c_2, \dots, c_k là các số thực, $c_k \neq 0$. Giả sử phương trình đặc trưng:

$$r^k - c_1 r^{k-1} - c_2 r^{k-2} - \dots - c_k = 0$$

có k nghiệm phân biệt r_1, r_2, \dots, r_k . Khi đó dãy số $\{a_n\}$ là nghiệm của công thức:

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k}$$

khi và chỉ khi

$$a_n = \alpha_1 r_1^n + \alpha_2 r_2^n + \dots + \alpha_k r_k^n$$

với $n = 0, 1, 2, \dots$, trong đó $\alpha_1, \alpha_2, \dots, \alpha_k$ là các hằng số

Ví dụ 4

Tìm nghiệm của công thức đệ quy

$$a_n = 6 a_{n-1} - 11 a_{n-2} + 6 a_{n-3}$$

với điều kiện đầu

$$a_0 = 2, \quad a_1 = 5, \quad a_2 = 15.$$

Giải: Phương trình đặc trưng

$$r^3 - 6 r^2 + 11 r - 6 = 0$$

có 3 nghiệm $r_1 = 1, \quad r_2 = 2, \quad r_3 = 3$.

Vì vậy, nghiệm có dạng

$$a_n = \alpha_1 1^n + \alpha_2 2^n + \alpha_3 3^n.$$

$$\text{CTĐQ: } a_n = c_1 a_{n-1} + \dots + c_k a_{n-k}$$

$$\text{Phương trình đặc trưng: } r^k - c_1 r^{k-1} - \dots - c_k = 0$$

Ví dụ 4

Sử dụng các điều kiện đầu ta có hệ phương trình sau đây để xác định các hằng số $\alpha_1, \alpha_2, \alpha_3$:

$$a_0 = 2 = \alpha_1 + \alpha_2 + \alpha_3$$

$$a_1 = 5 = \alpha_1 + \alpha_2 \cdot 2 + \alpha_3 \cdot 3$$

$$a_2 = 15 = \alpha_1 + \alpha_2 \cdot 4 + \alpha_3 \cdot 9.$$

Giải hệ phương trình trên ta thu được

$$\alpha_1 = 1, \alpha_2 = -1 \text{ và } \alpha_3 = 2.$$

Vậy nghiệm của công thức đã cho là

$$a_n = 1 - 2^n + 2 \cdot 3^n$$

Trường hợp tổng quát

Định lý 4. Cho c_1, c_2, \dots, c_k là các hằng số thực, với $c_k \neq 0$.

Giả sử PTĐT: $r^k - c_1 r^{k-1} - c_2 r^{k-2} - \dots - c_k = 0$

$$r^k - \sum_{i=1}^k c_i r^{k-i} = 0$$

có t nghiệm phân biệt r_1, r_2, \dots, r_t với bội tương ứng là m_1, \dots, m_t (với $m_1 + \dots + m_t = k$).

Khi đó, dãy $\{a_n\}$ là nghiệm của CTĐQ TTTN hệ số hằng

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \dots + c_k a_{n-k}$$

$$a_n = \sum_{i=1}^k c_i a_{n-i}$$

khi và chỉ khi

$$\begin{aligned} a_n = & (\alpha_{1,0} + \alpha_{1,1}n + \dots + \alpha_{1,m_1-1}n^{m_1-1}) r_1^n + \\ & (\alpha_{2,0} + \alpha_{2,1}n + \dots + \alpha_{2,m_2-1}n^{m_2-1}) r_2^n + \dots \\ & (\alpha_{t,0} + \alpha_{t,1}n + \dots + \alpha_{t,m_t-1}n^{m_t-1}) r_t^n \end{aligned}$$

$$a_n = \sum_{i=1}^t \left(\sum_{j=0}^{m_i-1} \alpha_{i,j} n^j \right) r_i^n$$

với $n=0,1,2,\dots$ và $\alpha_{i,j}$ là các hằng số $1 \leq i \leq t$ và

$0 \leq j \leq m_i - 1$ dựa vào các điều kiện đầu

Ví dụ 5

Giải công thức đệ quy:

$$c_n = -4c_{n-1} + 3c_{n-2} + 18c_{n-3}, \quad n \geq 3,$$

$$c_0 = 1; c_1 = 2; c_2 = 13.$$

Giải: Phương trình đặc trưng

$$r^3 + 4r^2 - 3r - 18 = (r - 2)(r + 3)^2 = 0$$

Vậy nghiệm tổng quát của công thức:

1.6. Giải công thức đệ quy: Các phương pháp khác

- Trên thực tế, khi phân tích độ phức tạp của một thuật toán nào đó nhờ sử dụng công thức đệ quy tuyến tính, thì công thức đệ quy ít khi có bậc lớn hơn 2.
- Do đó, ta cũng có thể sử dụng hai phương pháp sau đây để giải công thức đệ quy
 - Thay thế quay lui: xuất phát từ công thức đã cho, ta thế lần lượt lùi về các số hạng phía trước của công thức đệ quy
 - Cây đệ quy: biểu diễn công thức đệ quy bởi một cây đệ quy. Xuất phát từ công thức đệ quy đã cho, ta biểu diễn các số hạng đệ quy có kích thước dữ liệu đầu vào lớn ở mức A nào đó trên cây bởi các dữ liệu đầu vào nhỏ hơn ở mức $A+1$ của vậy, và tính các số hạng không đệ quy. Cuối cùng, tính tổng tất cả các số hạng không đệ quy ở tất cả các mức của cây.

1.6. Giải công thức đệ quy: Phương pháp 2: Thay thế quay lui

Ví dụ 1: Giải công thức đệ quy

$$T(n) = T(n-1) + 2n$$

$$\text{với } T(1) = 5$$

Giải:

Ta tiến hành thay thế lần lượt các số hạng của công thức đệ quy bằng cách lùi lại các số hạng phía trước:

$$\begin{aligned} T(n) &= T(n-1) + 2n \\ &= T((n-1)-1) + 2(n-1) + 2n \\ &= T(n-2) + 2(n-1) + 2n \\ &= T((n-2)-1) + 2(n-2) + 2(n-1) + 2n \\ &= T(n-3) + 2(n-2) + 2(n-1) + 2n \end{aligned}$$

1.6. Giải công thức đệ quy: Phương pháp 2: Thay thế quay lui

- Tiếp tục thay thế lùi ta được

$$\begin{aligned}T(n) &= T(n-3) + 2(n-2) + 2(n-1) + 2n \\ &= T(n-4) + 2(n-3) + 2(n-2) + 2(n-1) + 2n\end{aligned}$$

.....

$$= T(n-i) + \sum_{j=0}^{i-1} 2(n-j) \quad \text{giá trị hàm tại bước thay thế thứ } i$$

- Giải công thức tổng $\sum_{j=0}^{i-1} 2(n-j)$ ta được

$$\begin{aligned}T(n) &= T(n-i) + 2n(i-1) - 2(i-1)(i-1+1)/2 + 2n \\ &= T(n-i) + 2n(i-1) - i^2 + i + 2n \quad (1)\end{aligned}$$

- Giờ ta muốn loại bỏ số hạng $T(n-i)$. Để làm được điều này, tại bước lặp thay thế thứ bao nhiêu ta thu được điều kiện đầu $T(1)$?

Nhận thấy thu được $T(1)$ khi $n - i = 1$, tức là $i = n - 1$

- Thay thế vào biểu thức (1) ta có

$$\begin{aligned}T(n) &= T(1) + 2n(n-1-1) - (n-1)^2 + (n-1) + 2n \\ &= 5 + 2n(n-2) - (n^2-2n+1) + (n-1) + 2n = n^2 + n + 3\end{aligned}$$

1.6. Giải công thức đệ quy: Phương pháp 2: Thay thế quay lui

Ví dụ 2: Giải công thức đệ quy

$$T(n) = T(n/2) + c$$

với $T(1) = 2$, và c là hằng số

$$\begin{aligned} T(n) &= T(n/2) + c && \text{thay thế } T(n/2) \\ &= T(n/4) + c + c && \text{thay thế } T(n/4) \\ &= T(n/8) + c + c + c \\ &= T(n/2^3) + 3c \\ &= \dots \\ &= T(n/2^k) + kc \end{aligned}$$

$$\begin{aligned} T(n) &= T(n/2^{\log n}) + c \log n && \text{“chọn } k = \log n\text{”} \\ &= T(n/n) + c \log n \\ &= T(1) + c \log n = 2 + c \log n \end{aligned}$$

Ví dụ 3

Hàm tính giai thừa

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ n \cdot (n - 1)! & \text{if } n > 1 \end{cases}$$

Xét thuật toán đệ quy tính $n!$

Algorithm (FACTORIAL)

```
INPUT      :  $n \in \mathbb{N}$ 
OUTPUT     :  $n!$ 
1 IF  $n = 1$  THEN
2   return 1
3 END
4 ELSE
5   return FACTORIAL( $n - 1$ )  $\times$   $n$ 
6 END
```

Ví dụ 3

Factorial(n);

Algorithm (FACTORIAL)

```
INPUT      :  $n \in \mathbb{N}$ 
OUTPUT     :  $n!$ 
1 IF  $n = 1$  THEN
2   return 1
3 END
4 ELSE
5   return FACTORIAL( $n - 1$ )  $\times$   $n$ 
6 END
```

Bao nhiêu lần hàm Factorial được gọi khi chúng ta thực hiện lệnh **Factorial(n);** ?

- Khi $n = 1$, hàm Factorial được gọi 1 lần $\rightarrow T(1) = 1$
- Khi $n > 1$:
 - Gọi hàm Factorial 1 lần,
 - Cộng với số lần gọi trong lệnh đệ quy Factorial($n - 1$) $\rightarrow T(n-1)$

$$\left. \begin{array}{l} \text{Gọi hàm Factorial 1 lần,} \\ \text{Cộng với số lần gọi trong lệnh đệ quy Factorial}(n-1) \end{array} \right\} T(n) = 1 + T(n-1)$$

Do đó ta có công thức đệ quy:

$$T(1) = 1$$

$$T(n) = 1 + T(n - 1), n > 1$$

Tìm **công thức dạng hiện** tính $T(n)$ với mọi giá trị của n

1.6. Giải công thức đệ quy: Phương pháp 2: Thay thế quay lui

Factorial(n);

Algorithm (FACTORIAL)


```
INPUT      :  $n \in \mathbb{N}$ 
OUTPUT     :  $n!$ 
1 IF  $n = 1$  THEN
2   return 1
3 END
4 ELSE
5   return FACTORIAL( $n - 1$ )  $\times$   $n$ 
6 END
```

$$\begin{aligned} T(n) &= T(n-1) + 1, n > 1 \\ &= [T(n-2) + 1] + 1 \quad [\text{có 2 số "1"}] \\ &= [[T(n-3) + 1] + 1] + 1 \quad [\text{có 3 số "1"}] \\ &= \dots \\ &= T(n - (n-1)) + (n-1) \quad [\text{có } n-1 \text{ số "1"}] \\ &= T(1) + (n-1) \\ &= 1 + (n-1) \\ &= n \end{aligned}$$

Ví dụ 4

Algorithm (FACTORIAL)

```
INPUT      :  $n \in \mathbb{N}$ 
OUTPUT     :  $n!$ 
1 IF  $n = 1$  THEN
2   return 1
3 END
4 ELSE
5   return FACTORIAL( $n - 1$ )  $\times$   $n$ 
6 END
```



Có bao nhiêu phép nhân $M(n)$ được thực thi khi thực hiện hàm Factorial?

- Khi $n = 1$ không phép nhân nào được thực hiện $\rightarrow M(1) = 0$
- Khi $n > 1$:
 - Ta thực hiện 1 lần phép nhân.
 - Cộng với số lần thực hiện phép nhân trong lệnh gọi đệ quy Factorial($n - 1$) $\rightarrow M(n-1)$

Do đó ta có công thức đệ quy:

$$M(0) = 0; M(1) = 0$$

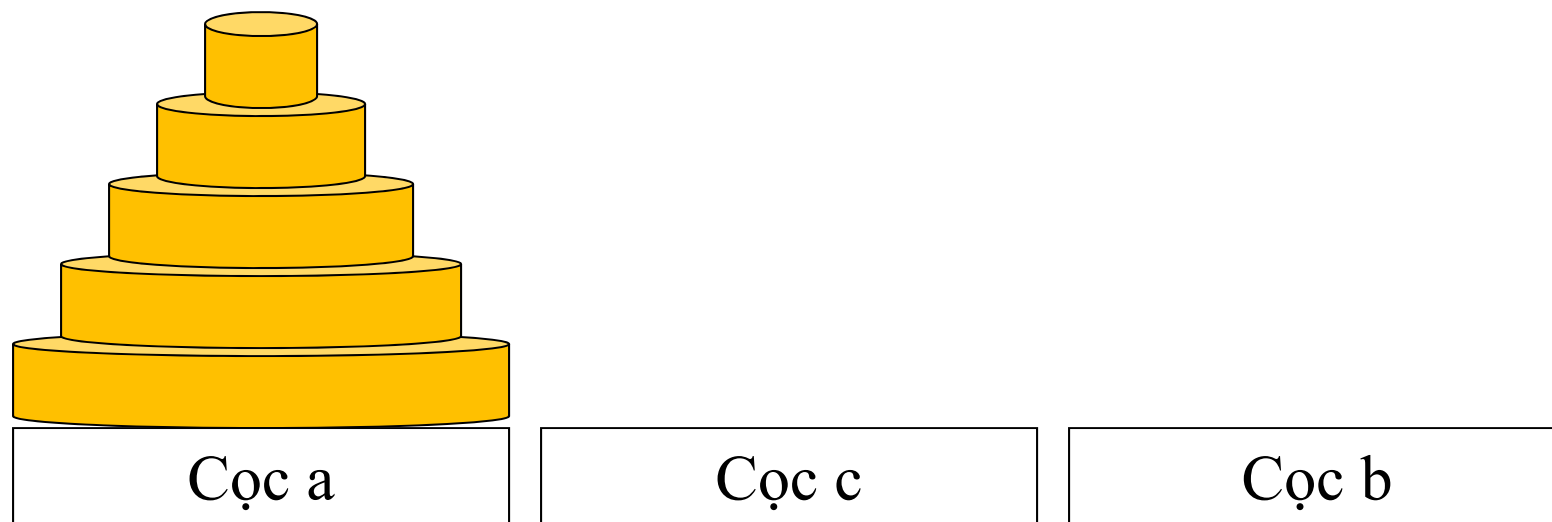
$$M(n) = 1 + M(n - 1), n > 1$$

1.6. Giải công thức đệ quy: Phương pháp 2: Thay thế quay lui

Ví dụ 5. (Bài toán tháp Hà nội). Trò chơi tháp Hà nội được trình bày như sau: “Có 3 cọc a, b, c . Trên cọc a có một chồng gồm n cái đĩa đường kính giảm dần từ dưới lên trên. Cần phải chuyển chồng đĩa từ cọc a sang cọc c tuân thủ qui tắc:

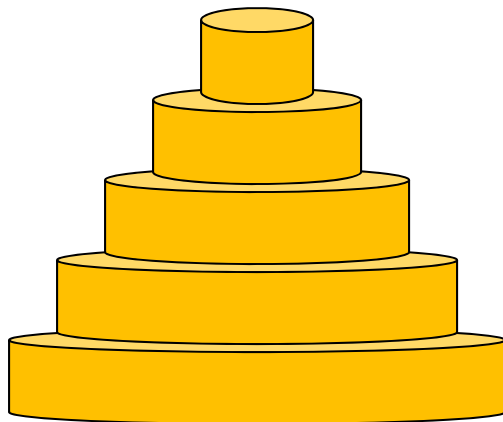
1. Mỗi lần chỉ chuyển 1 đĩa
2. Chỉ được xếp đĩa có đường kính nhỏ hơn lên trên đĩa có đường kính lớn hơn.
Trong quá trình chuyển được phép dùng cọc b làm cọc trung gian.

Bài toán đặt ra là: Tìm công thức đệ quy cho h_n là số lần di chuyển đĩa ít nhất cần thực hiện để hoàn thành nhiệm vụ đặt ra trong trò chơi tháp Hà nội.



Tower of Hanoi: $n=5$

1. *Mỗi lần chỉ chuyển 1 đĩa*
2. *Chỉ được xếp đĩa có đường kính nhỏ hơn lên trên đĩa có đường kính lớn hơn*



Cọc a

Cọc c

Cọc b

Bài toán: chuyển n đĩa từ cọc a sang cọc c sử dụng cọc b làm trung gian
Cần tìm h_n là số lần di chuyển đĩa ít nhất cần thực hiện

$$h_n = 2h_{n-1} + 1, n \geq 2.$$

Việc di chuyển đĩa gồm 3 bước:

(1) Chuyển $n-1$ đĩa từ cọc a đến cọc b sử dụng cọc c làm trung gian.

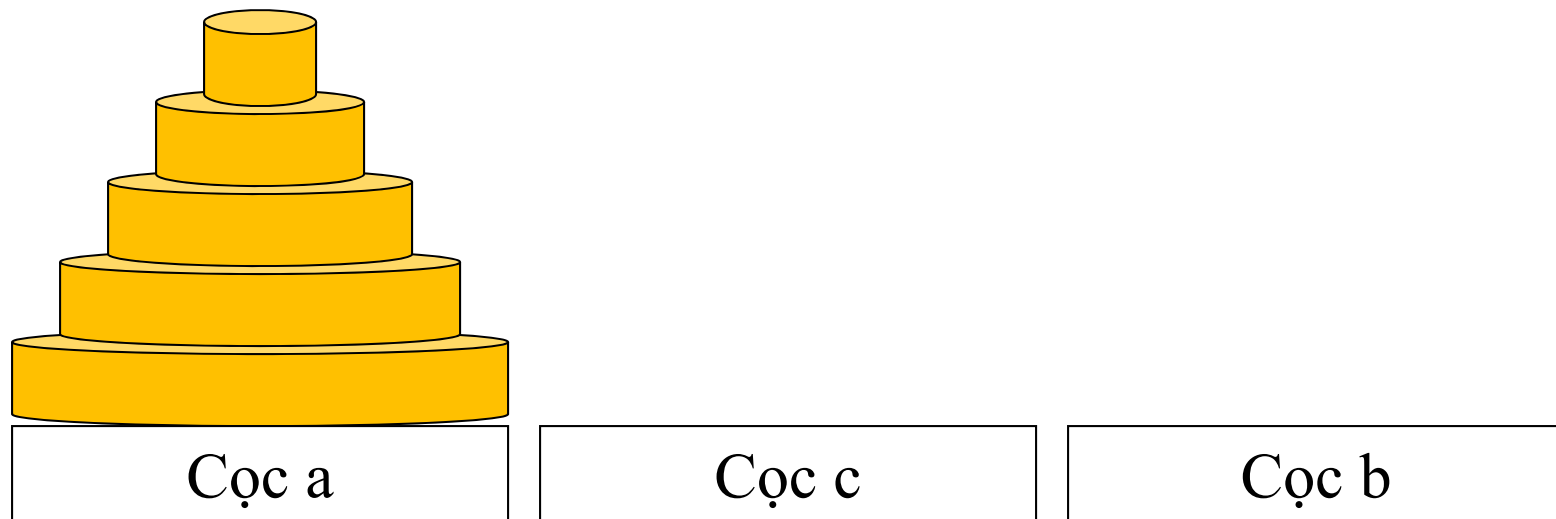
Bài toán kích thước $n-1 \rightarrow$ Số lần di chuyển $= h_{n-1}$

(2) Chuyển 1 đĩa (đĩa với đường kính lớn nhất) từ cọc a đến cọc c .

\rightarrow Số lần di chuyển $= 1$

(3) Chuyển $n-1$ đĩa từ cọc b đến cọc c (sử dụng cọc a làm trung gian).

Bài toán kích thước $n-1 \rightarrow$ Số lần di chuyển $= h_{n-1}$



Tower of Hanoi: $n=5$

(1) Chuyển $n-1$ đĩa từ cọc a đến cọc b sử dụng cọc c làm trung gian.

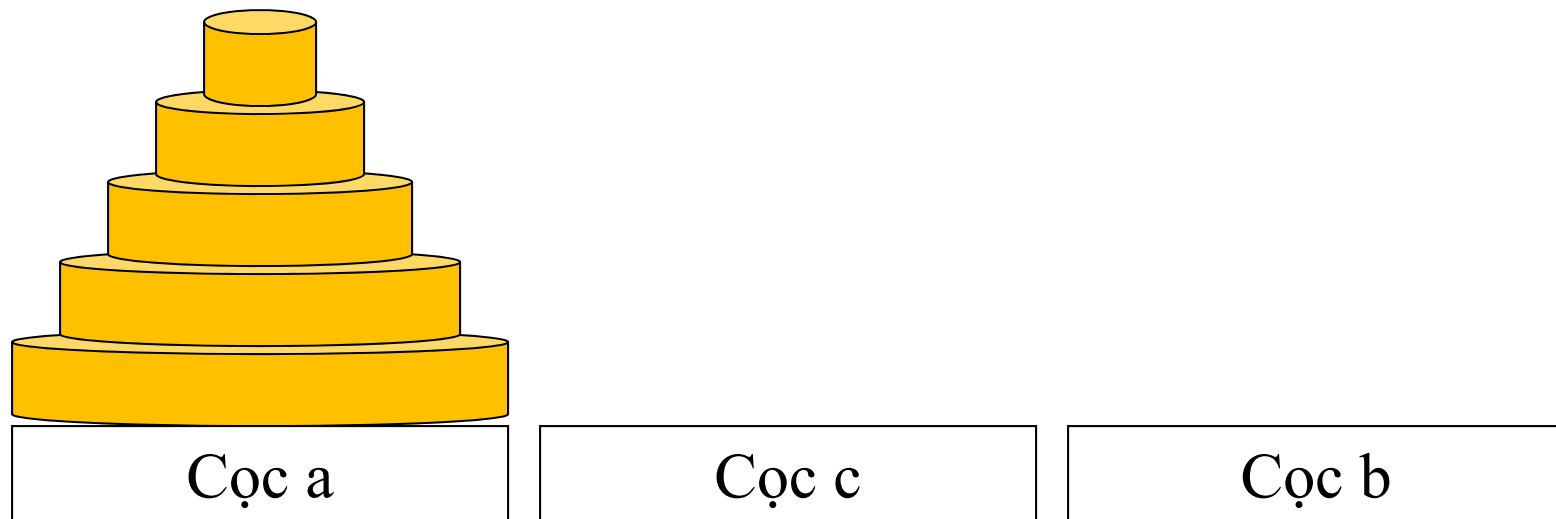
Bài toán kích thước $n-1 \rightarrow$ Số lần di chuyển $= h_{n-1}$

(2) Chuyển 1 đĩa (đĩa với đường kính lớn nhất) từ cọc a đến cọc c .

\rightarrow Số lần di chuyển $= 1$

(3) Chuyển $n-1$ đĩa từ cọc b đến cọc c (sử dụng cọc a làm trung gian).

Bài toán kích thước $n-1 \rightarrow$ Số lần di chuyển $= h_{n-1}$



1.6. Giải công thức đệ quy: Phương pháp 2: Thay thế quay lui

Ta có thể tìm được công thức trực tiếp cho h_n bằng phương pháp thế quay lui:

$$\begin{aligned}h_n &= 2 h_{n-1} + 1 \\&= 2 (2 h_{n-2} + 1) + 1 = 2^2 h_{n-2} + 2 + 1 \\&= 2^2 (2 h_{n-3} + 1) + 2 + 1 = 2^3 h_{n-3} + 2^2 + 2 + 1 \\&\dots \\&= 2^{n-1} h_1 + 2^{n-2} + \dots + 2 + 1 \\&= 2^{n-1} + 2^{n-2} + \dots + 2 + 1 \quad (\text{do } h_1 = 1) \\&= 2^n - 1\end{aligned}$$

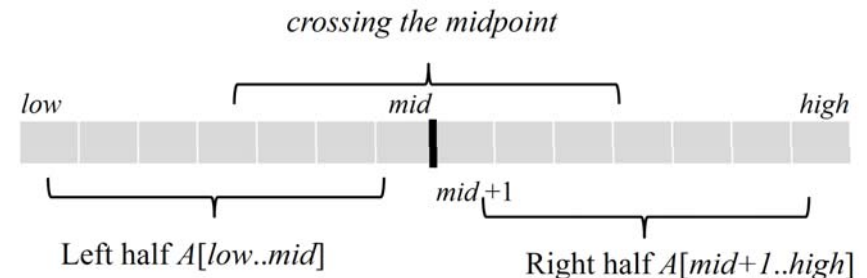
1.6. Giải công thức đệ quy: Phương pháp 3: Cây đệ quy

- Để giải công thức đệ quy bằng phương pháp này: ta sẽ vẽ cây đệ quy mô tả công thức đã cho
- Mỗi nút của cây tương ứng với 1 hàm dữ liệu đầu vào. Càng đi xuống phía dưới cây, kích thước dữ liệu đầu vào càng giảm.
- Mức cuối cùng của cây tương ứng với kích thước dữ liệu đầu vào nhỏ nhất

Ví dụ 1: Thuật toán đệ quy giải bài toán dãy con lớn nhất

```
MaxLeft(a, low, mid);
{
    maxSum = -∞; sum = 0;
    for (int k=mid; k>=low; k--) {
        sum = sum+a[k];
        maxSum = max(sum, maxSum);
    }
    return maxSum;
}
```

← $O(n)$



```
MaxRight(a, mid, high);
{
    maxSum = -∞; sum = 0;
    for (int k=mid; k<=high; k++){
        sum = sum+a[k];
        maxSum = max(sum, maxSum);
    }
    return maxSum;
}
```

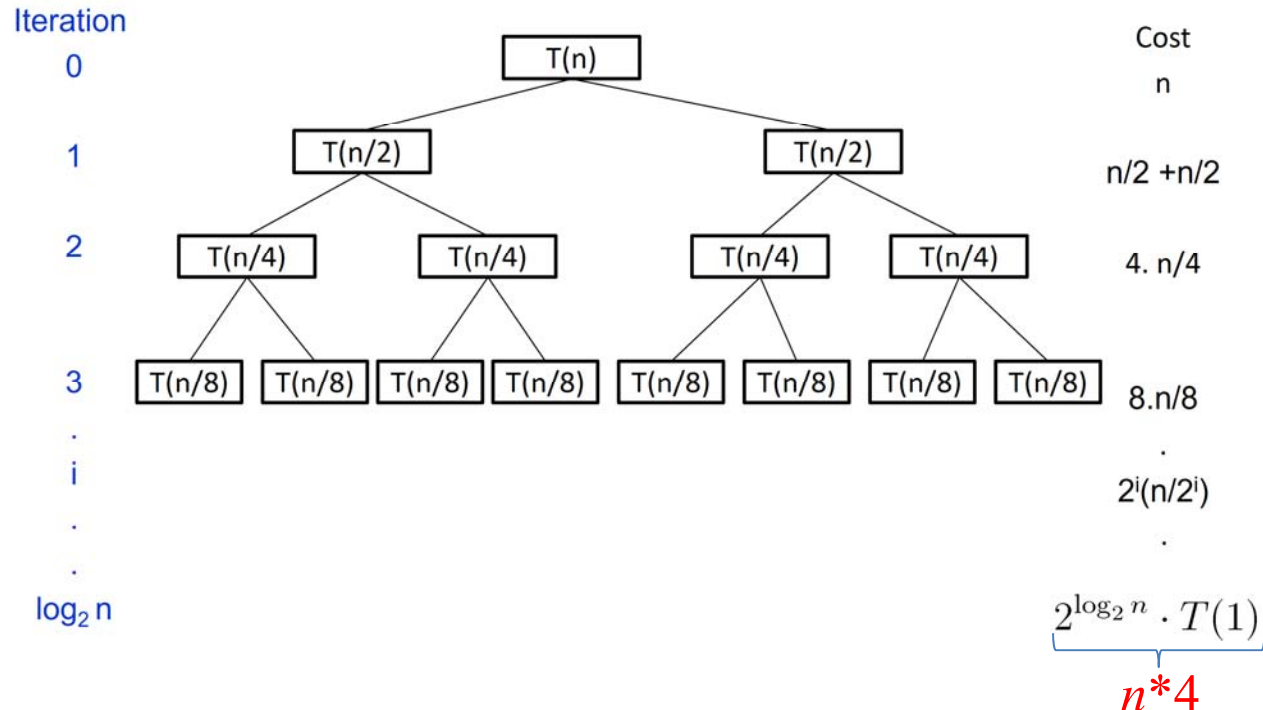
← $O(n)$

```
MaxSub(a, low, high);
{
    if (low = high) return a[low] //base case: only 1 element
    else
    {
        mid = (low+high)/2;
        wL = MaxSub(a, low, mid); ←  $T(n/2)$ 
        wR = MaxSub(a, mid+1, high); ←  $T(n/2)$ 
        wM = MaxLeft(a, low, mid) + MaxRight(a, mid, high);
        return max(wL, wR, wM);
    }
}
```

$T(n) = 2T(n/2) + O(n)$
 Giải bằng phương pháp cây đệ
 quy, ta sẽ thu được:
 → $T(n) = O(n \log n)$

Ví dụ 1: Giải công thức đệ quy:

$$T(n) = 2T(n/2) + n, \quad T(1) = 4$$



- Giá trị hàm $T(n)$ bằng tổng các giá trị tại tất cả các mức:

$$T(n) = \sum_{i=0}^{\log_2 n} 2^i \left(\frac{n}{2^i} \right)$$

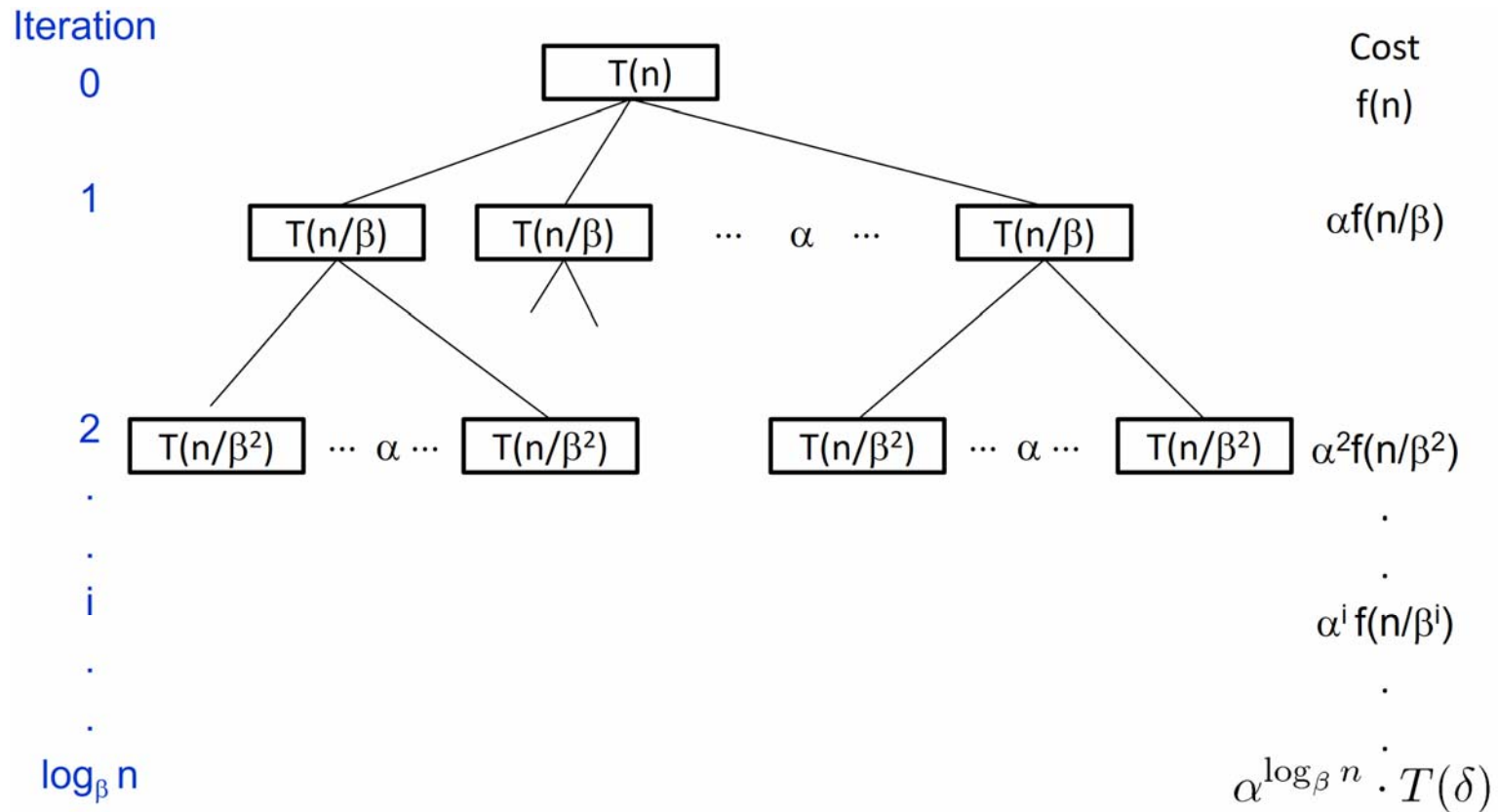
- Vì mức cuối (sâu nhất) $\log_2 n$ có giá trị $= 4n$, ta có

$$T(n) = 4n + \sum_{i=0}^{(\log_2 n - 1)} 2^i \frac{n}{2^i} = n(\log n) + 4n$$

1.6. Giải công thức đệ quy: Phương pháp 3: Cây đệ quy

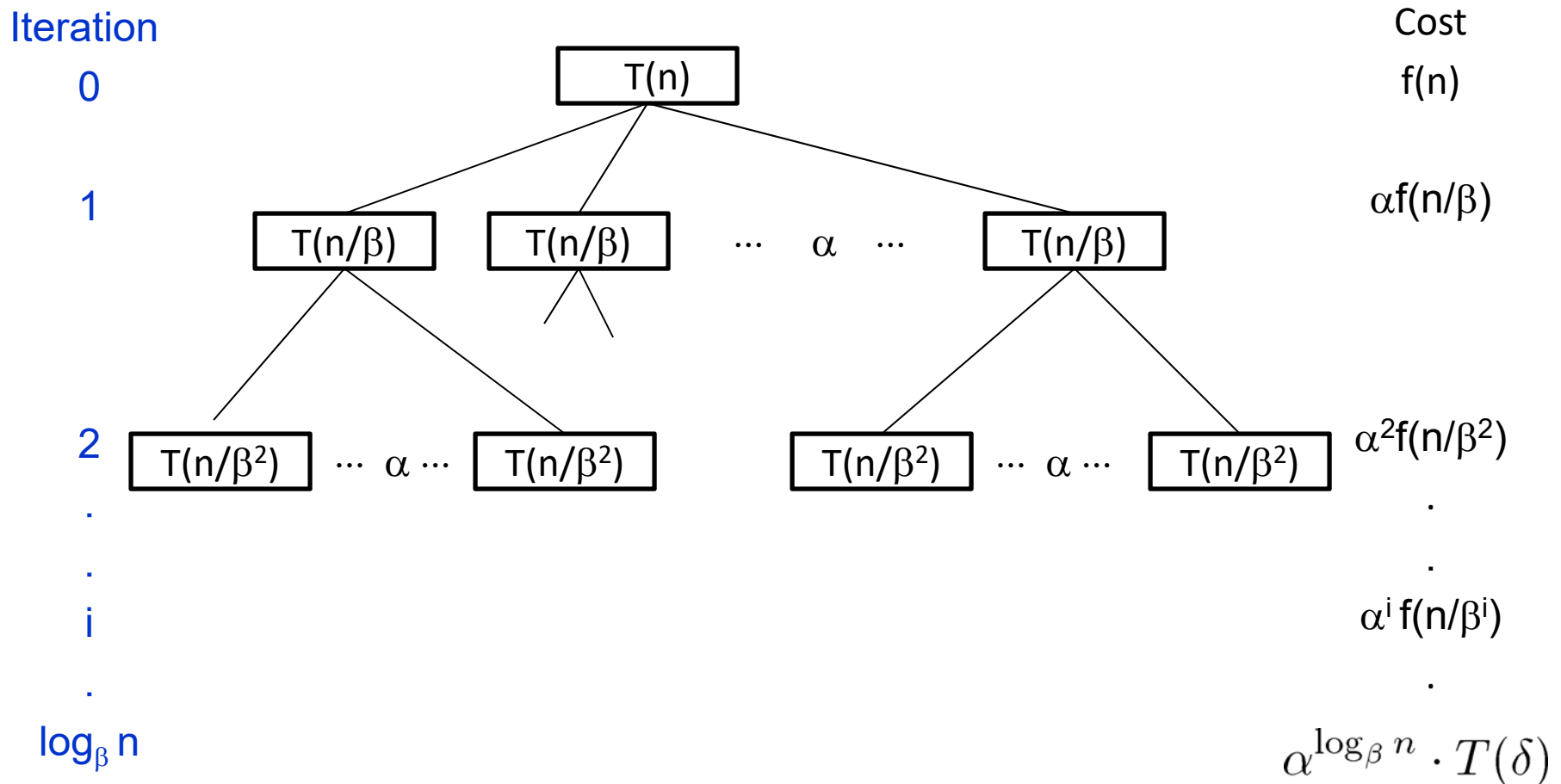
- Ví dụ 2: Giải công thức đệ quy

$$T(n) = \alpha T(n/\beta) + f(n), \quad T(\delta) = c$$



Ví dụ 2: Giải công thức đệ quy

$T(n) = \alpha T(n/\beta) + f(n), \quad T(\delta) = c$



- Giá trị của hàm bằng tổng các giá trị trên tất cả các mức của cây:

$$T(n) = \sum_{i=0}^{\log_{\beta} n} \alpha^i f\left(\frac{n}{\beta^i}\right) \quad \longrightarrow \quad T(n) = \alpha^{\log_{\beta} n} T(\delta) + \sum_{i=0}^{(\log_{\beta} n)-1} \alpha^i f\left(\frac{n}{\beta^i}\right)$$

Ví dụ 3: Tìm kiếm nhị phân dưới dạng đệ quy

Đầu vào: Mảng S gồm n phần tử: $S[0], \dots, S[n-1]$ đã sắp xếp theo thứ tự tăng dần; Giá trị **key**.

Đầu ra: chỉ số của phần tử có giá trị **key** nếu có; -1 nếu **key** không xuất hiện trong mảng S

```
int binsearch(int low, int high, int S[], int key)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (S[mid]==key) return mid;
        else if (key < S[mid])
            return binsearch(low, mid-1, S, key);
        else
            return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

Ví dụ 3: Tìm kiếm nhị phân dưới dạng đệ quy

```
int binsearch(int low, int high, int S[], int key)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (S[mid]==key) return mid;
        else if (key < S[mid])
            return binsearch(low, mid-1, S, key);
        else
            return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑														↑
lo														hi

Ví dụ 3: Tìm kiếm nhị phân dưới dạng đệ quy

```
int binsearch(int low, int high, int S[], int key)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (S[mid]==key) return mid;
        else if (key < S[mid])
            return binsearch(low, mid-1, S, key);
        else
            return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑							↑							↑
lo							mid							hi

Ví dụ 3: Tìm kiếm nhị phân dưới dạng đệ quy

```
int binsearch(int low, int high, int S[], int key)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (S[mid]==key) return mid;
        else if (key < S[mid])
            return binsearch(low, mid-1, S, key);
        else
            return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑						↑								
lo						hi								

Ví dụ 3: Tìm kiếm nhị phân dưới dạng đệ quy

```
int binsearch(int low, int high, int S[], int key)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (S[mid]==key) return mid;
        else if (key < S[mid])
            return binsearch(low, mid-1, S, key);
        else
            return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑			↑			↑								
lo			mid			hi								

Ví dụ 3: Tìm kiếm nhị phân dưới dạng đệ quy

```
int binsearch(int low, int high, int S[], int key)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (S[mid]==key) return mid;
        else if (key < S[mid])
            return binsearch(low, mid-1, S, key);
        else
            return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				↑		↑								
				lo		hi								

Ví dụ 3: Tìm kiếm nhị phân dưới dạng đệ quy

```
int binsearch(int low, int high, int S[], int key)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (S[mid]==key) return mid;
        else if (key < S[mid])
            return binsearch(low, mid-1, S, key);
        else
            return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				↑	↑	↑								
				lo	mid	hi								

Ví dụ 3: Tìm kiếm nhị phân dưới dạng đệ quy

```
int binsearch(int low, int high, int S[], int key)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (S[mid]==key) return mid;
        else if (key < S[mid])
            return binsearch(low, mid-1, S, key);
        else
            return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

↑
lo
hi

Ví dụ 3: Tìm kiếm nhị phân dưới dạng đệ quy

```
int binsearch(int low, int high, int S[], int key)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (S[mid]==key) return mid;
        else if (key < S[mid])
            return binsearch(low, mid-1, S, key);
        else
            return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

↑
lo
hi
mid

Ví dụ 3: Tìm kiếm nhị phân dưới dạng đệ quy

```
int binsearch(int low, int high, int S[], int key)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (S[mid]==key) return mid;
        else if (key < S[mid])
            return binsearch(low, mid-1, S, key);
        else
            return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33

key=31??

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

↑
lo
hi
mid

Ví dụ 3: Tìm kiếm nhị phân dưới dạng đệ quy

Đầu vào: Mảng S gồm n phần tử: $S[0], \dots, S[n-1]$ đã sắp xếp theo thứ tự tăng dần; Giá trị **key**.

Đầu ra: chỉ số của phần tử có giá trị **key** nếu có; -1 nếu **key** không xuất hiện trong mảng S

```
int binsearch(int low, int high, int S[], int key)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (S[mid]==key) return mid;
        else if (key < S[mid])
            return binsearch(low, mid-1, S, key);
        else
            return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

→ `binsearch(0, $n-1$, S , key);`

Có bao nhiêu lần hàm `binsearch` được gọi trong trường hợp tồi nhất ?

Ví dụ 3: Tìm kiếm nhị phân dưới dạng đệ quy

```
int binsearch(int low, int high, int S[], int key)
```

```
{
```

```
    if (low <= high)
```

```
    {
```

```
        mid = (low + high) / 2;
```

```
        if (S[mid]==key) return mid;
```

```
        else if (key < S[mid])
```

```
            return binsearch(low, mid-1, S, key);
```

```
        else
```

```
            return binsearch(mid+1, high, S, key);
```

```
    }
```

```
    else return -1;
```

```
}
```

→ $T(0) = ?$

→ $\text{binsearch}(0, -1, S, \text{key});$

→ $T(1) = ?$

→ $\text{binsearch}(0, 0, S, \text{key});$

→ $\text{binsearch}(0, -1, S, \text{key});$

→ $\text{binsearch}(1, 0, S, \text{key});$

→ $\text{binsearch}(0, n-1, S, \text{key});$

Gọi $T(n)$: số lần hàm `binsearch` được gọi trong trường hợp tồi nhất khi mảng S có n phần tử

- $T(0) = 1$
- $T(1) = 2$
- $T(2) = T(1) + 1 = 3$
- $T(4) = T(2) + 1 = 4$
- $T(8) = T(4) + 1 = 4 + 1 = 5$
- $T(n) = T(n/2) + 1$

Công thức đệ quy:

$$T(n) = T(n/2) + 1$$

$$T(0) = 1$$

$$T(1) = 2$$

Bài tập: Hãy giải công thức đệ quy này