ANALYSIS OF ALGORITHMS

Lecturer: Doctor Bùi Thanh Hùng
Director of Data Analytics & Artificial Intelligence Laboratory
Director of the Master/Undergraduate Information Systems Programme
Engineering Technology School
Thu Dau Mot University
Email: hungbt3@fe.edu.vn
Website: https://sites.google.com/site/hungthanhbui1980/

Why study algorithms?

Theoretical importance
- The cornerstone of computer science

Practical importance
- a practitioner's toolkit of known algorithms
- frameworks for designing and analyzing algorithms for new problems

Program = Data Structure + Algorithm

Major Algorithm Design Techniques/Strategies

- Brute force
- Decrease and conquer
- Divide and conquer
- Transform and conquer
- Space-time tradeoff
- Greedy approach
- Dynamic programming
- Iterative improvement
- Backtracking
- Branch and Bound

Analysis of Algorithms

- Difficulties with comparing programs instead of algorithms
  - How are the algorithms coded?
  - Which compiler is used?
  - What computer should you use?
  - What data should the programs use?
- Algorithm analysis should be independent of
  - Specific implementations
  - Compilers and their optimizers
  - Computers
  - Data

Analysis of Algorithms

- How good is the algorithm?
  - correctness (accuracy for approximation alg.)
  - time efficiency
  - space efficiency
  - optimality

- Approaches:
  - empirical (experimental) analysis
  - theoretical (mathematical) analysis

Theoretical analysis of time efficiency

Time efficiency is analyzed by determining the number of times the algorithm's *basic operation* is executed as a function of *input size*

- *Input size*: number of input items or, if matters, their size
- *Basic operation*: the operation contributing the most toward the running time of the algorithm:

    + - * /

    Phép gán

    So sánh

    Vòng lặp

    Số lần gọi đệ quy

Asymptotic order of growth

A way to classify functions according to their order of growth

- *practical* way to deal with complexity functions
- ignores constant factors and small input sizes

- Big-O
  - $O(g(n))$: class of functions $f(n)$ that grow ***no faster*** than $g(n)$

- Big-Theta
  - $\Theta(g(n))$: class of functions $f(n)$ that grow ***at same rate*** as $g(n)$

- Big-Omega
  - $\Omega(g(n))$: class of functions $f(n)$ that grow ***at least as fast*** as $g(n)$

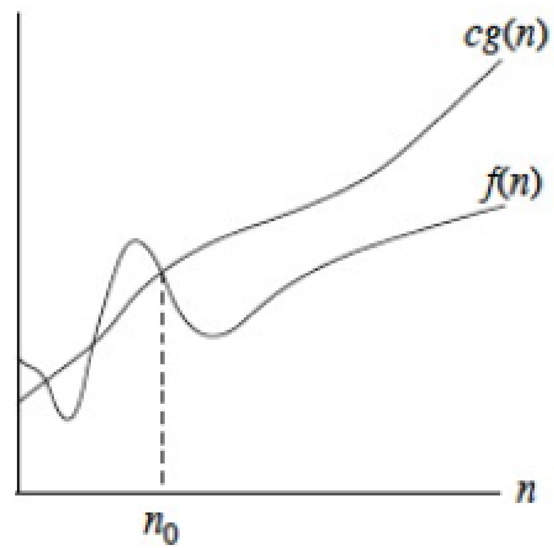
Big-O (asymptotic ≤)

Definition: $f(n)$ is in $O(g(n))$ if order of growth of $f(n) \le$ order of growth of $g(n)$ (within constant multiple),
i.e., there exist positive constant c and non-negative integer n_0 such that

$$f(n) \le c\, g(n) \text{ for every } n \ge n_0$$

Examples:

- $10n^2$ is $O(n^2)$
- $10n$ is $O(n^2)$
- $5n+20$ is $O(n)$



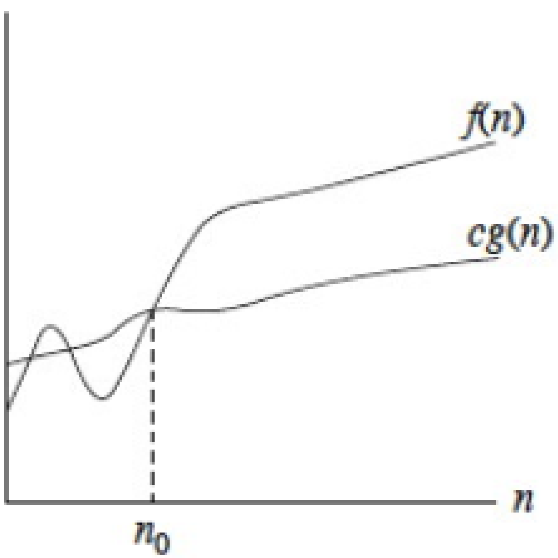
Ω (Omega, asymptotic ≥)

Definition: $f(n)$ is in $\Omega(g(n))$ if there exist positive constant c and non-negative integer n_0 such that

$$f(n) \geq c\, g(n) \text{ for every } n \geq n_0$$

These are all $\Omega(n^2)$:

- n^2
- $n^2 + 100n$
- $1000n^2 - 1000\,n$
- n^3



These are not:

- $n^{1.999}$
- n
- $\lg n$

Θ (Theta, asymptotic =)

Definition: $f(n)$ is in $\Theta(g(n))$ if there exist positive constants c_1, c_2 and non-negative integer n_0 such that

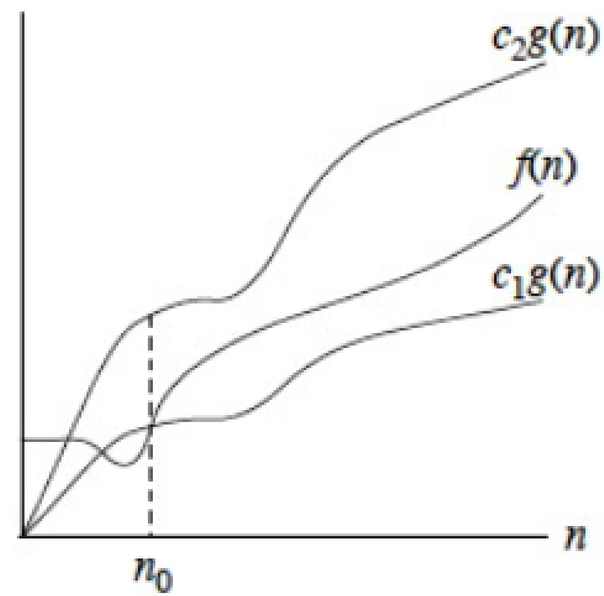
$$c_1\, g(n) \leq f(n) \leq c_2\, g(n) \text{ for every } n \geq n_0$$

Example:

- n^2-$2n$ is $\Theta(n^2)$
  - pick $c_1 = 0.5$, $c_2 = 1$, $n_0 = 4$

Find a tight Θ-bound for:

- $4n^3$
- $4n^3+2n$



Θ (Theta, asymptotic =)

Definition: $f(n)$ is in $\Theta(g(n))$ if there exist positive constants c_1, c_2 and non-negative integer n_0 such that

$$c_1\, g(n) \le f(n) \le c_2\, g(n) \text{ for every } n \ge n_0$$

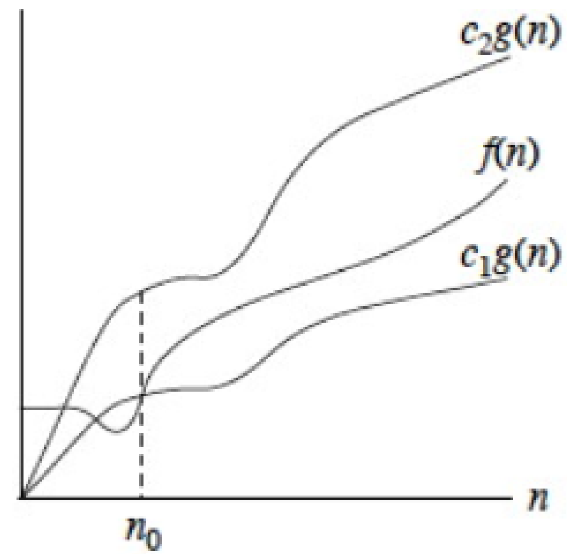
Example:

- n^2-$2n$ is $\Theta(n^2)$

$$\frac{n^2}{2} \le n^2 - 2n \le n^2$$

True for $n \geq 4$

  - pick $c_1 = 0.5$, $c_2 = 1$, $n_0 = 4$



Find a tight Θ-bound for:

- $4n^3$

$$4n^3 \le 4n^3 \le 4n^3$$

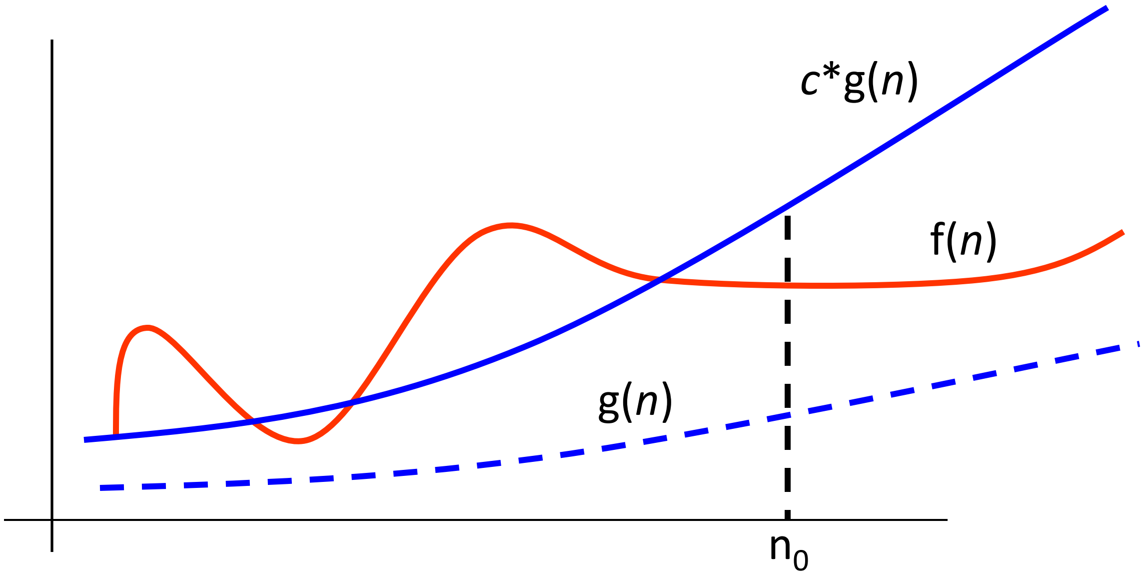
- $4n^3$+$2n$

$$4n^3 \le 4n^3 + 2n \le 6n^3$$

Big O notation

- Given a function f(n), we say g(n) is an (asymptotic) upper bound of f(n), denoted as f(n) = O(g(n)), if there exist a constant $c > 0$, and a positive integer n_0 such that f(n) $\leq$ c*g(n) for all $n \geq n_0$.

- f(n) is said to be bounded from above by g(n).
- O() is called the "big O" notation.



Growth Terms

- The most common growth terms can be ordered as follows: (note: many others are not shown)

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < \dots$

Note:

- "log" = log base 2, or $\log_2$; "$\log_{10}$" = log base 10; "ln" = log base e. In big O, all these log functions are the same.

Order-of-Magnitude Analysis and Big O Notation

(a)

Function	n = 10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Figure - Comparison of growth-rate functions in tabular form

Order-of-Magnitude Analysis and Big O Notation

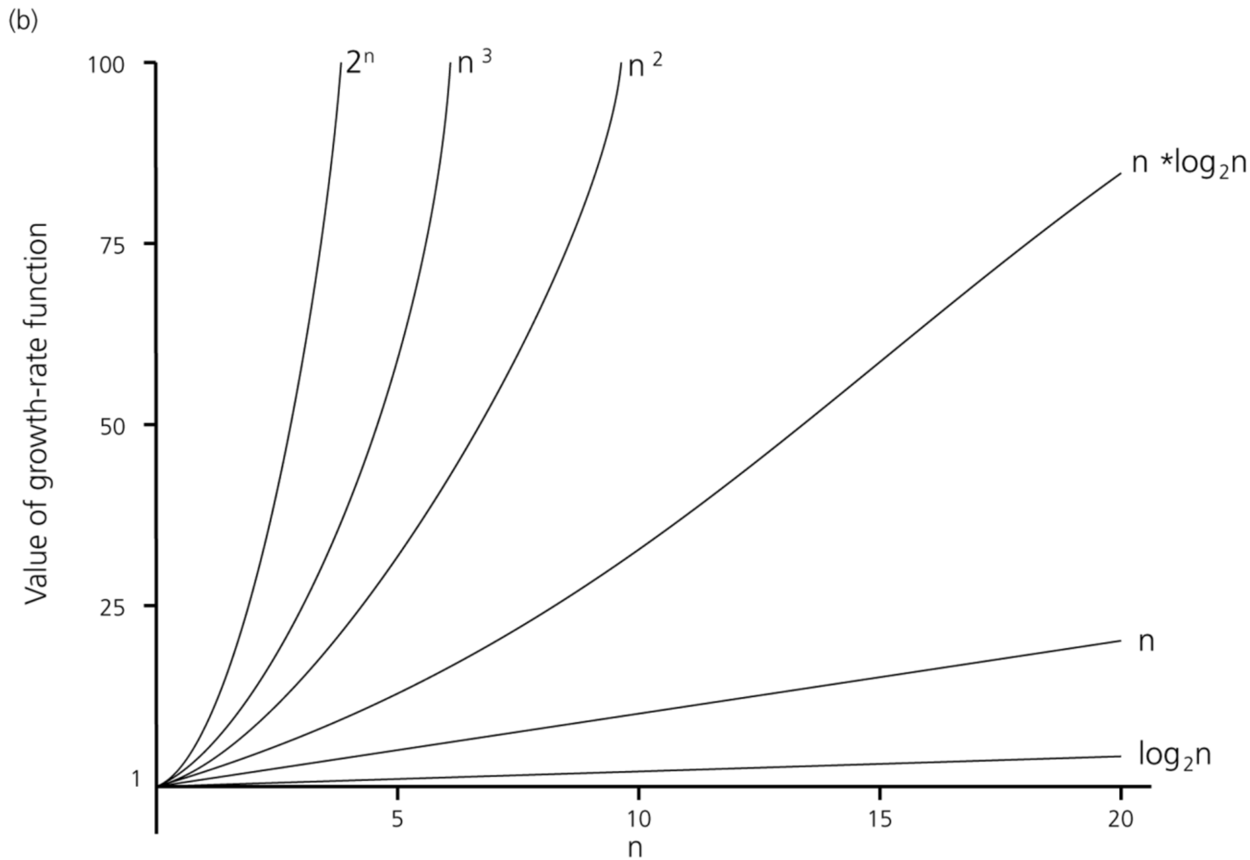


Figure - Comparison of growth-rate functions in graphical form

Some rules of thumb and examples

- Basically just count the number of statements executed.
- If there are only a small number of simple statements in a program – $O(1)$
- If there is a 'for' loop dictated by a loop index that goes up to n – $O(n)$
- If there is a nested 'for' loop with outer one controlled by n and the inner one controlled by m – $O(n*m)$
- For a loop with a range of values n, and each iteration reduces the range by a fixed constant fraction (eg: ½) – $O(\log n)$
- For a recursive method, each call is usually $O(1)$. So
  - if n calls are made – $O(n)$
  - if $n \log n$ calls are made – $O(n \log n)$

Example

- Image that we have the number of calculations is S(n)
- $S(n) = 1 + 2 + ... + n$

- What is complexity?

Example

- S(n) = 1 + 2+…. + n < n+n+ …+ n = n^2
- O(n^2)

Example

$$\Sigma_{1 \le i \le n}\, i = 1+2+\ldots+n = n(n+1)/2 \;\approx\; n^2/2 \in \Theta(n^2)$$

Example

$\Sigma_{l \leq i \leq u} 1 =$

In particular, $\Sigma_{1 \leq i \leq n} 1 =$

Example

$\Sigma_{l \le i \le u} 1 = 1+1+\ldots+1 = u - l + 1$

In particular, $\Sigma_{1 \le i \le n}\, 1 = n\text{-}1+1 = n \in \Theta(n)$

Example

$$\Sigma_{1 \leq i \leq n} i^2 = 1^2 + 2^2 + \ldots + n^2 =$$

Example

$$\Sigma_{1\le i\le n}\, i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$$

Example

$\Sigma_{0 \le i \le n} a^i = a^0 + a^1 + \ldots + a^n =$

In particular, $\Sigma_{0 \le i \le n} 2^i =$

Example

$\Sigma_{0 \leq i \leq n} a^i = a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1) \in \Theta(a^n)$

In particular, $\Sigma_{0 \leq i \leq n} 2^i = 2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

Example

$$\Sigma_{1 \le i \le n} 1/i = 1/1 + 1/2 + \ldots + 1/n$$

Example

$$\Sigma_{1 \le i \le n} 1/i = 1/1 + 1/2 + \ldots + 1/n \approx \ln n + 0.5772\ldots \in \Theta(\log n)$$

Example

$$\Sigma_{1 \le i \le n} \lg i = \lg 1 + \lg 2 + \ldots + \lg n$$

Example

$$\Sigma_{1 \le i \le n} \lg i = \lg 1 + \lg 2 + \ldots + \lg n \in \Theta(n \log n)$$

$\Sigma_{l \le i \le u} 1 = 1+1+\ldots+1 = u - l + 1$

In particular, $\Sigma_{1 \le i \le n} 1 = n\text{-}1+1 = n \in \Theta(n)$

$\Sigma_{1 \le i \le n}\, i = 1+2+\ldots+n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

$\Sigma_{1 \le i \le n}\, i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

$\Sigma_{0 \le i \le n}\, a^i = a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1) \in \Theta(a^n)$

In particular, $\Sigma_{0 \le i \le n}\, 2^i = 2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

$\Sigma_{1 \le i \le n}\, 1/i = 1/1+1/2+\ldots+1/n \approx \ln n + 0.5772\ldots \in \Theta(\log n)$

$\Sigma_{1 \le i \le n} \lg i = \lg 1+\lg 2+\ldots+\lg n \in \Theta(n\log n)$

$$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma\, b_i$$

$$\Sigma c a_i = c\Sigma a_i$$

$$\Sigma_{l \le i \le u} a_i = \Sigma_{l \le i \le m} a_i + \Sigma_{m+1 \le i \le u} a_i$$

$$\Sigma_{l \le i \le u} (a_i - a_{i-1}) = a_u - a_{l-1}$$

Approximation by definite integrals

$$\int_{l-1}^{u} f(x)\mathrm{d}x \le \Sigma_{l \le i \le u} f(i) \le \int_{l}^{u+1} f(x)\mathrm{d}x \quad \text{for nondecreasing } f(x)$$

$$\int_{l}^{u+1} f(x)\mathrm{d}x \le \Sigma_{l \le i \le u} f(i) \le \int_{l-1}^{u} f(x)\mathrm{d}x \quad \text{for nonincreasing } f(x)$$

Example

- f(n)= n log (n!) + (3 n^2 +2 n)log n.

Logarit

1	$\log_a 1 = 0$
2	$\log_a a = 1$
3	$\log_a a^M = M$
4	$a^{\log_a N} = N$
5	$\log_a (N_1.N_2) = \log_a N_1 + \log_a N_2$
6	$\log_a (\frac{N_1}{N_2}) = \log_a N_1 - \log_a N_2$

7	$\log_a N^\alpha = \alpha.\log_a N$
8	$\log_a N^2 = 2.\log_a \lvert N \rvert$
9	$\log_a N = \log_a b.\log_b N$
10	$\log_b N = \dfrac{\log_a N}{\log_a b}$
11	$\log_a b = \dfrac{1}{\log_b a}$
12	$\log_{a^\alpha} N = \dfrac{1}{\alpha}\log_a N$
13	$a^{\log_b c} = c^{\log_b a}$

Example

- f(n)= n log (n!) + (3 n^2 +2 n)log n.

- log(n!) = O(n log n)
- n log (n!) = O(n^2 log n)
- (3 n^2 +2 n) = O(n^2)
- (3 n^2 +2 n)log n = O(n^2 log n)

- O(n^2 log n)

Example

- f(n) = (n+3) log (n^2 +4) + 5 n^2

Example

- f(n) = (n+3) log (n^2 +4) + 5 n^2

- n+3= O(n)
- log (n^2+4)=O(log n).
- n>2 log(n^2 +4) < log(2 n^2) < log 2 + log n^2 = log 2 + 2log n < 3 log n.
- (n+3) log (n^2 +4) = O(nlog n).
- 5 n^2 = O(n^2).
- f(n) = O(max { nlog n, n^2 }) = O(n^2).

Example

- f(x) = 2^x + 23

Example

- f(x) = 2^x + 23

- x > 5 ta có f(x) < 2 × 2^x
- f(x) = O(2^x).
- 2^x < f(x) với mọi x>0.
- O(2^x) là đánh giá tốt nhất đối với f(x) (hay nói cách khác 2^x là cùng bậc với f(x)).

Example

```
int sum = 0;
for (int i=1; i<n; i=i*2)
{
  sum++;
}
```

Example

```
int sum = 0;
for (int i=1; i<n; i=i*2) {
   sum++;
}
```

- It is clear that sum is incremented only when

  i = 1, 2, 4, 8, …, 2^k where $k = \lfloor \log_2 n \rfloor$

  There are k+1 iterations. So the complexity is O(k) or O(log n)

Note:

- In Computer Science, log n means $\log_2 n$.
- When 2 is replaced by 10 in the 'for' loop, the complexity is O($\log_{10} n$) which is the same as O($\log_2 n$).
- $\log_{10} n = \log_2 n \,/\, \log_2 10$

Example

let's assume that *n* is some power of 3

```
int sum = 0;
for (int i=1; i<n; i=i*3)
{
  for (j=1; j<=i; j++) {
    sum++;
  }
}
```

Example

let's assume that *n* is some power of 3

```
int sum = 0;
for (int i=1; i<n; i=i*3) {
   for (j=1; j<=i; j++) {
      sum++;
   }
}
```

- $f(n) = 1 + 3 + 9 + 27 + \ldots + 3^{(\log_3 n)}$

  $= 1 + 3 + \ldots + n/9 + n/3 + n$

  $= n + n/3 + n/9 + \ldots + 3 + 1$ (reversing the terms in previous step)

  $= n * (1 + 1/3 + 1/9 + \ldots)$

  $\leq n * (3/2)$

  $= 3n/2$

  $= O(n)$

Example

Work out the computational complexity of the following piece of code.

```
for ( i=1; i < n; i *= 2 ) {
        for ( j = n; j > 0; j /= 2 ) {
                for ( k = j; k < n; k += 2 ) {
                        sum += (i + j * k );
                }
        }
}
```

Example

Work out the computational complexity of the following piece of code.

```
for ( i=1; i < n; i *= 2 ) {
        for ( j = n; j > 0; j /= 2 ) {
                for ( k = j; k < n; k += 2 ) {
                        sum += (i + j * k );
                }
        }
}
```

Running time of the inner, middle, and outer loop is proportional to n, log n, and log n, respectively. Thus the overall Big-Oh complexity is $O(n(\log n)^2)$.

Example

```
def: BinarySearch(el, a):
        l=0
        r= len(a)-1
        m= a[(l+r)//2]
        if el < a[m]:
          el > a[m]:
          el=a[m]

a=[4,5,7,1,3,9,12]
a=sorted(a)
BinarySearch(9,a)
```

Input
Comparation
Bestcase
Worstcase
-> O()

Example

Xét độ phức tạp của thuật toán, giả thiết rằng có n= 2^k phần tử.

```
def  BinarySearch(x, a):
        first =0
        last =len(a)-1
        found =False
        while (first<=last and not found ):
                index= (first + last) // 2
                if (x == a[index]): found = True
                elif (x< a[index]): last = index –1
                else: first = index +1
        if (not found ): index = -1
        return index
a=[4,5,7,1,3,9,12]
a=sorted(a)
print(BinarySearch(9,a))
```

Example

Xét độ phức tạp của thuật toán, giả thiết rằng có n= 2^k phần tử.

```
def  BinarySearch(x, a):
        first =0
        last =len(a)-1
        found =False
        while (first<=last and not found ):
                index= (first + last) // 2
                if (x == a[index]): found = True
                elif (x< a[index]): last = index –1
                else: first = index +1
        if (not found ): index = -1
        return index
a=[4,5,7,1,3,9,12]
a=sorted(a)
print(BinarySearch(9,a))
```

- Số phép toán so sánh tối đa là 2k+1 = $2\log_2 n$.
- Hay độ phức tạp O(logn), độ phức tạp logarit.

Example

```
public static int USCLN(int a,int b){
        int x= a;
        int y=b;
        while (y>0) {
                int r = x % y;
                x = y;
                y = r;
        }
return x;
}
```

Example

```
public int void USCLN(int a,int b){
int x= a;
int y=b;
while (y>0) {
        int r = x % y;
        x = y;
        y = r
}
return x;
```

Định lý Lamé:
Cho a và b là các số nguyên dương với a >= b. Số phép chia cần thiết để tìm USCLN(a,b) nhỏ hơn hoặc bằng 5 lần số chữ số của b trong hệ thập phân (hay nói cách khác thuộc O($\log_2 b$) hay O(log *n*).

Analysis of Different Cases

Worst-Case Analysis

- Interested in the worst-case behaviour.
- A determination of the maximum amount of time that an algorithm requires to solve problems of size *n*

Best-Case Analysis

- Interested in the best-case behaviour
- Not useful

Average-Case Analysis

- A determination of the average amount of time that an algorithm requires to solve problems of size *n*
- Have to know the probability distribution
- The hardest

Divide-and-Conquer

The most-well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances

2. Solve smaller instances recursively

3. Obtain solution to original (larger) instance by combining these solutions

General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

<u>Master Theorem</u>: If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Note: The same results hold with O instead of Θ.

Examples: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$

$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$

$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$

General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

<u>Master Theorem</u>: If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Note: The same results hold with O instead of Θ.

Examples: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$

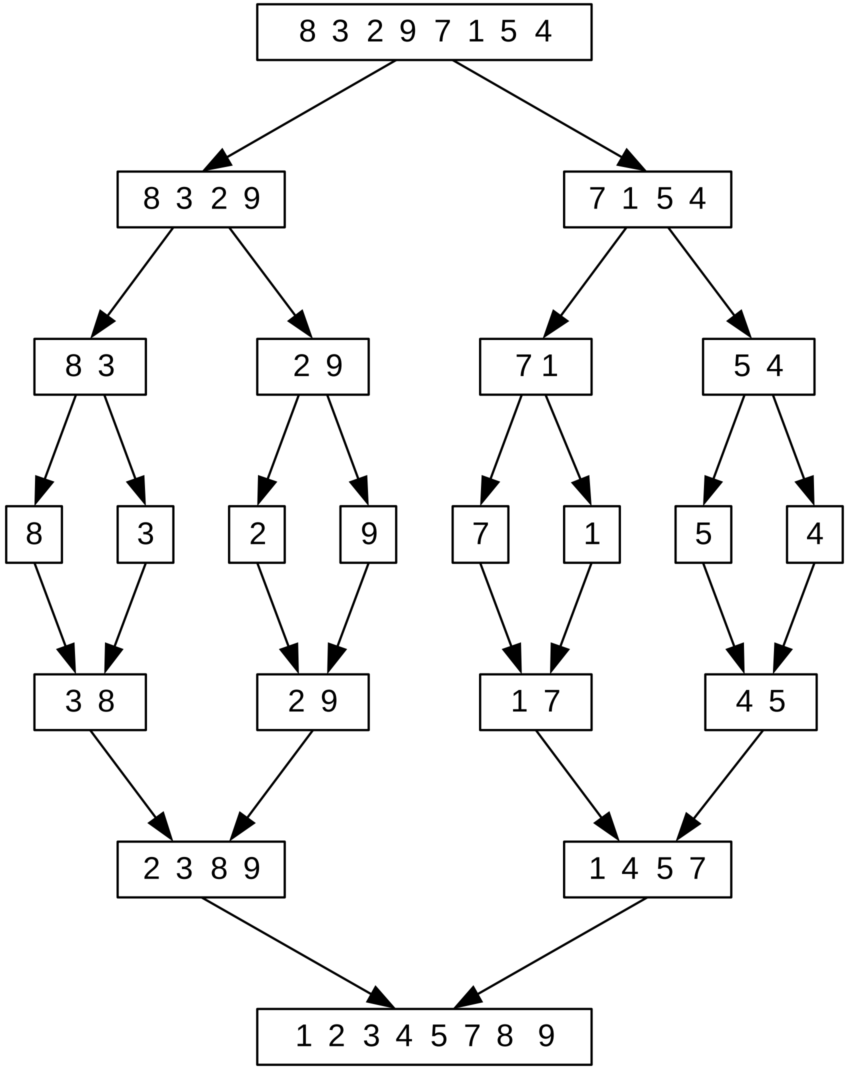
a= 4, b= 2, f(n) = Θ(n) = Θ(n^d) -> d= 1

Case 3 ($a > b^d$) $T(n) \in \Theta(n^{\log_2 4}) = \Theta(n\text{^}2)$

$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$

$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$

Mergesort Example



The non-recursive version of Mergesort starts from merging single elements into sorted pairs.

Mergersort

```
def mergeSort(a):
    if len(a) > 1:
        mid = len(a) // 2
        b = a[:mid]
        c = a[mid:]
        mergeSort(b)
        mergeSort(c)
        merge(b,c,a)
```

Analysis of Mergesort

- All cases have same efficiency: $\Theta(n \log n)$

  $T(n) = 2T(n/2) + \Theta(n)$, $T(1) = 0$

- Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting:

  $$\lceil \log_2 n! \rceil \approx n \log_2 n - 1.44n$$

- Space requirement: $\Theta(n)$ (<u>not</u> in-place)

- Can be implemented without recursion (bottom-up)

Ex1- Bài toán nhân 2 ma trận

- ✓ *Input size*
- ✓ *Basic operation*
- ✓ *Best case*
- ✓ *Worst case – summation for C(n)*

Ex2- Bài toán Gaussian elimination

Algorithm *GaussianElimination*($A[0..n-1, 0..n]$)

//Implements Gaussian elimination of an n-by-$(n+1)$ matrix A

for $i \leftarrow 0$ to $n - 2$ do
    for $j \leftarrow i + 1$ to $n - 1$ do
    for $k \leftarrow n$ downto i do
        $A[j,k] \leftarrow A[j,k] - A[i,k] * A[j,i] / A[i,i]$

Find the efficiency class and a constant factor improvement.

- ✓ *Input size*
- ✓ *Basic operation*
- ✓ *Best case*
- ✓ *Worst case – summation for C(n)*

Ex3- Bài toán cái túi

Có n đồ vật, vật thứ i có trọng lượng a[i] và giá trị c[i]. Hãy chọn ra một số các đồ vật, mỗi vật một cái để xếp vào 1 vali có trọng lượng tối đa V sao cho tổng giá trị của vali là lớn nhất.

1- Giải bài toán với mỗi vật chọn 1 lần

2- Giải bài toán với mỗi vật chọn n lần

- ✓ *Input size*
- ✓ *Basic operation*
- ✓ *Best case*
- ✓ *Worst case – summation for C(n)*