HUST

ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

ONE LOVE. ONE FUTURE.

DATA STRUCTURES AND ALGORITHMS

ĐẠI HỌC
BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY
OF SCIENCE AND TECHNOLOGY

DATA STRUCTURES AND ALGORITHMS

WEEK 1 : THE BASIC CONCEPTS

ONE LOVE. ONE FUTURE.

CONTENTS

OBJECTIVES

After this lesson, students can:

1. Understand some basic concepts about algorithm
2. Know to use asymptotic notation to analyze the complexity of an algorithm
3. Know how to analyze the complexity of an algorithm

CONTENTS

1. ILLUSTRATIVE EXAMPLE

- The maximum subarray problem:
  - Given an array of n numbers: $a_1, a_2, \dots, a_n$
  - The contiguous subarray $a_i, a_{i+1}, \dots, a_j$ with $1 \le i \le j \le n$ is a subarray of the given array and $\sum_{k=i}^{j} a_k$ *is called as the value of this subarray*
  - **The task is to find the maximum value of all possible subarrays**, *in other words, find the maximum* $\sum_{k=i}^{j} a_k$. The subarray with the maximum value is called as the maximum subarray.

Example: Given the array -2, **11, -4, 13**, -5, 2 then the maximum subarray is 11, -4, 13 with the value = 11+ (-4)+13 =20

1. ILLUSTRATIVE EXAMPLE

- Method 1: Brute force
  - Browse all possible subarray off the given array: a_i, a_{i+1}, …, a_j where $0 \le i \le j \le n\text{-}1$, and calculate the sum of all elements in subarray to find the maximum sum.

```
int maxSum = a[0];
for (int i = 0; i <= n-1; i++) {
   for (int j = i; j <= n-1; j++) {
        int sum = 0;
        for (int k = i; k <= j; k++)  sum += a[k];
        if (sum > maxSum)   maxSum = sum;
    }
}
```

1. ILLUSTRATIVE EXAMPLE

- Method 1: Brute force
  - Browse all possible subarray off the given array: a_i, a_{i+1} , …, a_j where $0 \le i \le j \le n\text{-}1$, and calculate the sum of all elements in subarray to find the maximum sum.
  - **Analyze the algorithm:** we count the number of time the statement **sum += a[k]** that the algorithm need to perform. The number is:

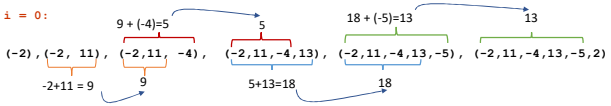
```
int maxSum = a[0];
for (int i = 0; i<=n-1; i++) {
   for (int j = i; j<=n-1; j++) {
       int sum = 0;
       for (int k=i; k<=j; k++)  sum += a[k];
       if (sum > maxSum)   maxSum = sum;
    }
}
```

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

1. ILLUSTRATIVE EXAMPLE

- Method 2: Brute force with better implementation

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



- We could see that, we can calculate the sum of the elements from position *i* to *j* from the sum of the elements from *i* to *j*-1 with just one addition:

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

The sum of elements from *i* to *j* — The sum of elements from *i* to *j*-1

1. ILLUSTRATIVE EXAMPLE

- Method 2: Brute force with better implementation

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

The sum of elements from *i* to *j* — The sum of elements from *i* to *j*-1

```
int maxSum = a[0];
for (int i=0; i<=n-1; i++) {
  for (int j=i; j<=n-1; 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-1; i++) {
   int sum = 0;
   for (int j=i; j<=n-1; j++) {
      sum += a[j];
      if (sum > maxSum) maxSum = sum;
   }
}
```


1. ILLUSTRATIVE EXAMPLE

- Method 2: Brute force with better implementation
  - **Analyze the algorithm:** We count the number of summation operations that the algorithm need to execute, it means count the number of times that the statement **sum += a[j]** is executed. The number of summation operations is:

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

```
int maxSum = a[0];
for (int i=0; i<=n-1; i++) {
   int sum = 0;
   for (int j=i; j<=n-1; j++) {
      sum += a[j];
      if (sum > maxSum) maxSum = sum;
   }
}
```

1. ILLUSTRATIVE EXAMPLE

- The number of times that the summation operations need to do is:
  - Method1. Brute force $\frac{n^3}{6}+\frac{n^2}{2}+\frac{n}{3}$
  - Method 2. Brute force with better implementation $\frac{n^2}{2}+\frac{n}{2}$
- For the same problem, we have proposed two algorithms that require different numbers of operations, and therefore will require different calculation times.
- The table below shows the calculation time of the above two algorithms, with the assumption: the computer can perform 10^8 additions per second.

Complexity	n=10	Time (sec)	n=100	Time (sec)	$n=10^4$	Time	$n=10^6$	Time
n^3	10^3	10^{-5}	10^6	10^{-2} sec	10^{12}	2.7 hours	10^{18}	115 days
n^2	100	10^{-6}	10000	10^{-4} sec	10^8	1 sec	10^{12}	2.7 hours

CONTENTS

2. BASIC CONCEPTS ABOUT ALGORITHM

- An algorithm for solving a given problem is a defined procedure that includes a finite sequence of steps that need to be performed to obtain an output from a given input (input) of the problem.

Input **Algorithm** **Output**

- Some basic characteristics of the algorithm:
  - Precision
  - Finiteness
  - Uniqueness
  - Generality

2. BASIC CONCEPTS ABOUT ALGORITHM

- Given 2 or more algorithms to solve the same problem, how do we select the best one?
- Some criteria for selecting an algorithm:
  1) Is it easy to implement, understand, modify?
  2) How long does it take to run it to completion? **TIME**
  3) How much of computer memory does it use? **SPACE**

2. BASIC CONCEPTS ABOUT ALGORITHM

- Complexity of an algorithm:
  - How to measure the computation time?
    - The computation time of an algorithm depends on the size of input data (size increases, then computation time increases).
    - Thus, analyze running time as a function of input size. However, in some cases, even on inputs of the same size, running time can be very different
    - Example: In order to find the first prime number in an array: the algorithm scans the array from left to right
      - Array 1: 3 9 8 12 15 20 (algorithm stops when considering the first element)
      - Array 2: 9 8 3 12 15 20 (algorithm stops when considering the 3rd element)
      - Array 3: 9 8 12 15 20 3 (algorithm stops when considering the last element)

      ➔ There are 3 types of computation time

2. BASIC CONCEPTS ABOUT ALGORITHM

- Types of computation time:

Best-case:

T(n) = minimum time of algorithm on any input of size n.

Average-case:

T(n) = expected time of algorithm over all inputs of size n.

Worst-case:

T(n) = maximum time of algorithm on any input of size n.

2. BASIC CONCEPTS ABOUT ALGORITHM

- There are two ways to evaluate the computation time:
  - **Experimental Evaluation of computation time:**
    - Write a program implementing the algorithm
    - Run the program with inputs of varying size and composition
    - Use a method like clock( ) to get an accurate measure of the actual running time

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

  - **Theory: use asymptotic notations**

CONTENTS

3. ASYMPTOTIC NOTATION

- Các ký hiệu tiệm cận (asymptotic notation):

$\Theta, \Omega, O, \omega$

  - 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.
  - Ví dụ, khi nói thời gian tính của thuật toán cỡ $\Theta(n^2)$, tức là, thời gian tính tỉ lệ thuận với n^2 cộng thêm các đa thức bậc thấp hơn.

3. ASYMPTOTIC NOTATION

- Asymptotic notation:

$\Theta, \Omega, O, \omega$

  - Used to describe the calculation time of an algorithm, describing the increase in runtime depending on the input data size.
  - For example, when saying the computation time is$\Theta(n^2)$, that is, the computation time is proportional to n^2 plus lower order terms.

3. ASYMPTOTIC NOTATION

3.1. Asymptotic notation theta Θ

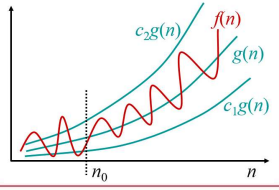
- For a given function $g(n)$, we denote by $\Theta(g(n))$ the set of functions:

$\Theta(g(n))$ = {$f(n)$: there exists constants c_1, c_2 and n_0 such that:

$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$, for all $n \ge n_0$}

(Set of all functions that have the same *rate of growth* as $g(n)$)

- When we say that one function is theta of another, we mean that neither function goes to infinity faster than the other.



3. ASYMPTOTIC NOTATION

3.1. Asymptotic notation theta Θ

- Example: Prove that $10n^2 - 3n = \Theta(n^2)$

We need to show with which values of the constants n_0, c_1, c_2 then the inequality in the definition of the theta notation is correct:

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

Suggestion: Make c_1 a little smaller than the leading (the highest) coefficient, and c_2 a little bigger.

➔ Select: $c_1 = 1$, $c_2 = 11$, $n_0 = 1$ then we have

$n^2 \le 10n^2 - 3n \le 11n^2$, *for* $n \ge 1$

➔ $\forall n \ge 1$: $10n^2 - 3n = \Theta(n^2)$

Note: For polynomial functions: To compare the growth rate, it is necessary to look at the term with the highest coefficient

3. ASYMPTOTIC NOTATION

3.2. Asymptotic notation big Oh O

- For a given function $g(n)$, we denote by $O(g(n))$ the set of functions:

$O(g(n)) = \{f(n)$: there exists positive constants c and n_0 such that:

$f(n) \le cg(n)$ for all $n \ge n_0\}$

(Set of all functions whose *rate of growth* is the same as or lower than that of $g(n)$)

- $O(g(n))$ is the set of functions that go to infinity no faster than $g(n)$.
- Example: Prove that $2\boldsymbol{n} + 10 = \boldsymbol{O(n)}$

➔ $f(n) = 2n+10$, $g(n) = n$

- Need to find constants c and n_0 such that:

$2\boldsymbol{n} + 10 \le \boldsymbol{cn}$ for all $n \ge n_0$

➔ $(\boldsymbol{c} - 2)\,\boldsymbol{n} \ge 10$

➔ $\boldsymbol{n} \ge 10/(\boldsymbol{c} - 2)$

➔ Select $\boldsymbol{c} = 3$ and $\boldsymbol{n_0} = 10$

3. ASYMPTOTIC NOTATION

3.2. Asymptotic notation big Oh O

- Note: $f(n) = 50n^3 + 20n + 4$ is $O(n^3)$

  Would be correct to say is $O(n^3+n)$

  Not useful, as n^3 exceeds by far n, for large values

  Would be correct to say is $O(n^5)$

  OK, but $g(n)$ should be as close as possible to $f(n)$
- Simple Rule: Drop lower order terms and constant factors
  - Example:
    - All these functions are $O(n)$: n, $3n$, $61n + 5$, $22n - 5$, ...
    - All these functions are $O(n^2)$: n^2, $9\,n^2$, $18\,n^2 + 4n - 53$, ...
    - All these functions are $O(n \log n)$: $n(\log n)$, $5n(\log 99n)$, $18 + (4n - 2)(\log(5n + 3))$,...

3. ASYMPTOTIC NOTATION

3.2. Asymptotic notation Omega Ω

- For a given function $g(n)$, we denote by $\boldsymbol{\Omega}(g(n))$ the set of functions:

$\Omega(g(n)) = \{f(n)$: there exists positive constants c and n_0 such that:

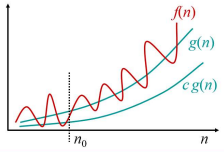
$cg(n) \le f(n)$ for all $n \ge n_0\}$

(Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$)

- $\Omega(g(n))$ is the set of functions that go to infinity no slower than $g(n)$
- Example: Prove that $5n^2 = \Omega(n)$

Need to find c and n_0 such that $\boldsymbol{cn} \le 5n^2$ for $n \ge n_0$

Inequality is correct when $c = 1$ and $n_0 = 1$



CONTENTS

4. ALGORITHMIC ANALYSIS TECHNIQUES

1. **Consecutive Statements**: The sum of running time of each segment.

Running time of "P; Q", where P is implemented first, then Q, is

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

or if using asymptotic Theta:

Time(*P*; *Q*) = Θ(max(*Time*(*P*), *Time*(*Q*)).

2. **FOR loop**: The number of iterations times the time of the inside statements.

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

Assume running time of *P*(*i*) is *t*(*i*), then the running time of for loop is $\sum_{i=1}^{m} t(i)$

3. **Nested loops**: The product of the number of iterations times the time of the inside statements.

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

Assume the running time of *P*(*j*) is *t*(*j*), then the running time of this nested loops is:

4. ALGORITHMIC ANALYSIS TECHNIQUES

4. **If/Else**

```
if (condition)
    S1;
else
    S2;
```

The testing time plus the larger running time of the S1 and S2.

4. ALGORITHMIC ANALYSIS TECHNIQUES

- Example

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

$O(n^2)$

$O(n^2)$

$O(n^2)$

4. ALGORITHMIC ANALYSIS TECHNIQUES

- **The characteristic statement** is the statement being executed with frequency at least as well as any statement in the algorithm.
- If the execution time of each statement is bounded above by a constant, then the running time of the algorithm will be the same size as the number of times the execution of the characteristic statement. Thus, in order to evaluate the running time, one can count the number of times the characteristic statement being executed
- Example 1: Function to calculate Fibonacci numbers f_0=0; f_1=1; f_n= f_{n-1} + f_{n-2}

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

The number of times this characteristic statement being executed is *n*

➔ The running time of Fibiter is *O*(*n*)

4. ALGORITHMIC ANALYSIS TECHNIQUES

- Example 2: The maximum subarray problem
  - Method 1: Brute force

```
int maxSum = a[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;
   }
}
```

➔ Select the statement **sum+=a[k]** as the characteristic statement
➔ Computation of the algorithm: $O(n^3)$

4. ALGORITHMIC ANALYSIS TECHNIQUES

- Example 2: The maximum subarray problem
  - Method 2: Brute force with better implementation

```
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;
   }
}
```

➔ Select the statement **sum+=a[k]** as the characteristic statement
➔ Computation of the algorithm: $O(n^2)$

4. ALGORITHMIC ANALYSIS TECHNIQUES

- Example 3: Give asymptotic big-Oh notation for the running time $T(n)$ of the following statement segment:

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

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


HUST

hust.edu.vn fb.com/dhbkhn

SUMMARY AND SUGGESTIONS

1. The lesson presented basic concepts of algorithms and algorithm complexity
2. Following this lesson, learners will learn about recursion - general diagram and some examples

THANK YOU !