

Introduction to Algorithms and Data Structures

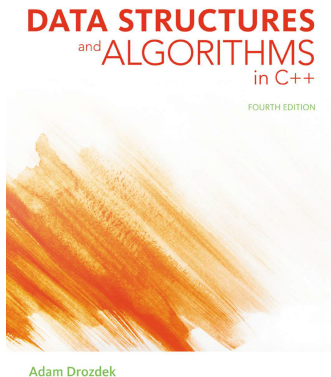
Doan Nhat Quang

doan-nhat.quang@usth.edu.vn
University of Science and Technology of Hanoi
ICT department

2018-2019

Course objectives:

- ▶ Provide basic knowledges about algorithms and data structures.
- ▶ Be able to choose appropriate data structures for a specifiqu problem.
- ▶ Approach different algorithms and solve a problem in informatics.



Recommended Textbook: Data Structures and Algorithms in C++
4th edition, Adam Drozdek

- ▶ Dev-C++: a free, portable, fast and simple C/C++ IDE
- ▶ Visual C++ Express: a free set of tools
- ▶ Online: <http://cpp.sh/> for simple programs



Why Study Algorithms

- ▶ Many problems can be solved by using a computer
 - ▶ want it to go faster? Process more data?
 - ▶ want it to do something that would otherwise be impossible?
- ▶ Technology improves many aspects but
 - ▶ it might be costly
 - ▶ good algorithmic design can do much better and might be cheaper
 - ▶ super-computers cannot handle a bad algorithm

Example

Suppose that a sequence of a_1, a_2, \dots, a_n ($n \geq 2$) is available, find the maximum?

Example

Suppose that a sequence of a_1, a_2, \dots, a_n ($n \geq 2$) is available, find the maximum?

- 1 Step 1: Given that $Max = a_1$
and the index $i = 2$,

Example

Suppose that a sequence of a_1, a_2, \dots, a_n ($n \geq 2$) is available, find the maximum?

- 1 Step 1: Given that $Max = a_1$
and the index $i = 2$,
- 2 Step 2: If $i > n$ then go to
Step 6

Example

Suppose that a sequence of a_1, a_2, \dots, a_n ($n \geq 2$) is available, find the maximum?

- 1 Step 1: Given that $Max = a_1$
and the index $i = 2$,
- 2 Step 2: If $i > n$ then go to
Step 6
- 3 Step 3: If $a_i > Max$ then
 $Max = a_i$

Example

Suppose that a sequence of a_1, a_2, \dots, a_n ($n \geq 2$) is available, find the maximum?

- ① Step 1: Given that $Max = a_1$
and the index $i = 2$,
- ② Step 2: If $i > n$ then go to
Step 6
- ③ Step 3: If $a_i > Max$ then
 $Max = a_i$
- ④ Step 4: Increment index i

Example

Suppose that a sequence of a_1, a_2, \dots, a_n ($n \geq 2$) is available, find the maximum?

- ① Step 1: Given that $Max = a_1$
and the index $i = 2$,
- ② Step 2: If $i > n$ then go to
Step 6
- ③ Step 3: If $a_i > Max$ then
 $Max = a_i$
- ④ Step 4: Increment index i
- ⑤ Step 5: Go to Step 2

Example

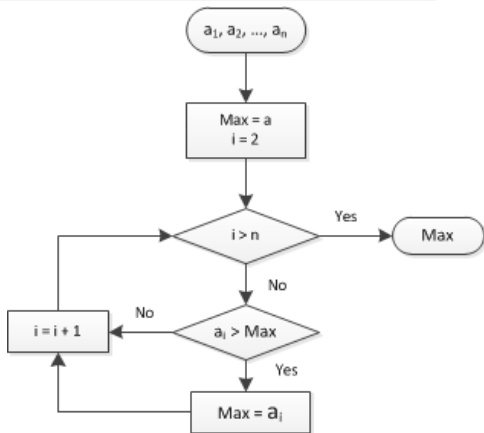
Suppose that a sequence of a_1, a_2, \dots, a_n ($n \geq 2$) is available, find the maximum?

- ① Step 1: Given that $Max = a_1$
and the index $i = 2$,
- ② Step 2: If $i > n$ then go to
Step 6
- ③ Step 3: If $a_i > Max$ then
 $Max = a_i$
- ④ Step 4: Increment index i
- ⑤ Step 5: Go to Step 2
- ⑥ Step 6: Return Max

Example

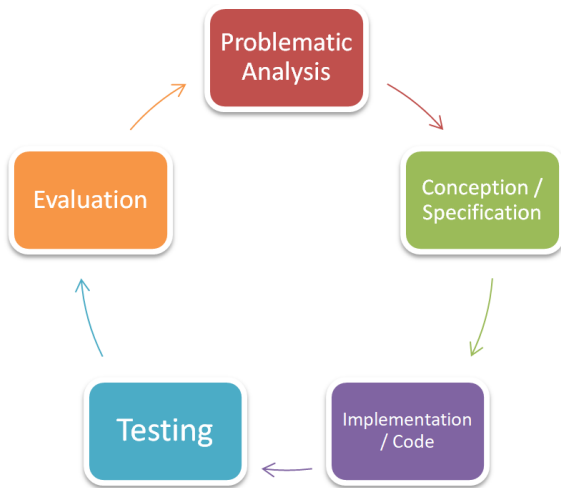
Suppose that a sequence of a_1, a_2, \dots, a_n ($n \geq 2$) is available, find the maximum?

- 1 Step 1: Given that $Max = a_1$ and the index $i = 2$,
- 2 Step 2: If $i > n$ then go to Step 6
- 3 Step 3: If $a_i > Max$ then $Max = a_i$
- 4 Step 4: Increment index i
- 5 Step 5: Go to Step 2
- 6 Step 6: Return Max



Algorithms and approaches

Strategy to solve a problem:



- ▶ Do you know any algorithm?

- ▶ Do you know any algorithm?
- ▶ The travelling salesman problem (Shortest Paths): "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?"
- ▶ Job scheduling: assign different tasks at particular times with various constraints.
- ▶ How to determine the best move for chess/go?

From Algorithms to Artificial Intelligence and Machine Learning

- ▶ Algorithms are the theory/core concepts of AI and Machine Learning
- ▶ Applications are widely large:
 - ▶ Banking: Fraud detection, Stock market analysis
 - ▶ Business: Online customer support, Virtual personal assistants
 - ▶ Health: Medical diagnosis, Medical image processing
 - ▶ Transport: Intelligent/smart vehicles, Transport optimization
 - ▶ etc.

Concept

From the original problem, we have to :

- ▶ identify the needs, the ideas of the problem
- ▶ find an approach, an appropriate algorithm to solve the problem

Data Structure

In this course, we will study basic structures:

- ▶ Arrays, Pointers
- ▶ Linked Lists, Stacks, Queues
- ▶ Tree, Binary Tree

Definition

An algorithm consists of a sequence of instructions that must be followed to solve a problem.

Definition







An algorithm consists of a sequence of instructions that must be followed to solve a problem.

Programming definition

An algorithm consists of a sequence of instructions in programming language to solve a problem on computer. Algorithms can be represented by flowchart, pseudo-code or code.

Flowcharts

Flowcharts are used in designing and documenting simple processes or programs. Flowcharts help visualize and understand a process

Shape	Name	Shape	Name
	Flow Line		Decision
	Terminal		Input/Output
	Process		Annotation

Pseudo-code

- ▶ Pseudo-code is a high-level description
- ▶ Pseudo-code is concrete and easy for human comprehension
- ▶ Pseudo-code is usual used to describe algorithms

Syntax

- ▶ Control flow:
 - ▶ if then.... (if else....)
 - ▶ while (...) ... do
 - ▶ repeat until (...)
 - ▶ for do....
- ▶ Method declaration
 - ▶ Input
 - ▶ Output

findMax (a)

```
1:  $Max = a_1$ 
2: for  $i = 2 \rightarrow n$  do
3:   if  $a_i > Max$  then
4:      $Max = a_i$ 
5:   end if
6: end for
7: return  $Max$ 
```

A good algorithm must possess the following properties:

A good algorithm must possess the following properties:

- ▶ **Finiteness** The algorithm must always terminate after a **finite number of steps**. **Stopping condition** should be asserted.

A good algorithm must possess the following properties:

- ▶ **Finiteness** The algorithm must always terminate after a **finite number of steps**. **Stopping condition** should be asserted.
- ▶ **Definiteness**: Each instruction must be precisely defined and concise. **Repetition** should be avoided at all cost.

A good algorithm must possess the following properties:

- ▶ **Finiteness** The algorithm must always terminate after a **finite number of steps**. **Stopping condition** should be asserted.
- ▶ **Definiteness**: Each instruction must be precisely defined and concise. **Repetition** should be avoided at all cost.
- ▶ **Input/Output**: An algorithm may or may not have **inputs** but an algorithm has one or more **ouputs** available when the algorithm terminates

A good algorithm must possess the following properties:

- ▶ **Finiteness** The algorithm must always terminate after a **finite number of steps**. **Stopping condition** should be asserted.
- ▶ **Definiteness**: Each instruction must be precisely defined and concise. **Repetition** should be avoided at all cost.
- ▶ **Input/Output**: An algorithm may or may not have **inputs** but an algorithm has one or more **ouputs** available when the algorithm terminates
- ▶ **Effectiveness**: All operations to be performed must be sufficiently basic that they can be done exactly and in finite length.

Criteria are often used to evaluate an algorithm:

- ▶ **Correctness** : does the algorithm always produce correct output for each given input? This asserts that the algorithm produces expected results.
- ▶ **Efficiency/Complexity**: An algorithm is always optimized to have low running time as possible (time complexity). During the programming, variables are declared and located in memory, an algorithm is always optimized to have low allocation memory as possible (space complexity).

Algorithm types we will consider include:

- ▶ Simple recursive algorithms
- ▶ Divide and conquer algorithms
- ▶ Dynamic programming algorithms
- ▶ Greedy algorithms
- ▶ Backtracking algorithms
- ▶ Randomized algorithms

A simple recursive algorithm:

- ▶ convert the main problem to sub-problems
- ▶ solve the base cases directly
- ▶ recur with a simpler sub-problem

Recursive Algorithms

A simple recursive algorithm:

- ▶ convert the main problem to sub-problems
- ▶ solve the base cases directly
- ▶ recur with a simpler sub-problem

Example: Fibonacci numbers 0, 1, 1, 2, 3, 5, 8, ...

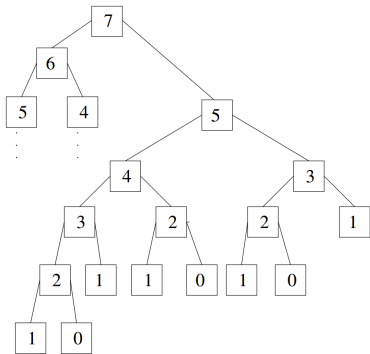
```
1  int fibo(int n){  
2      if ((n == 0) || (n == 1)) // base cases  
3          return n;  
4      return fibo(n-1) + fibo(n-2);  
5  }
```

Recursive Algorithms

The picture shows that the solution computes solutions to the subproblems more than once for no reason:

7	6	5	4	3	2	1	0
1	1	2	3	5	8	13	21

→ Complexity is exponential,
 $O(2^n)$



For Hanoi Tower problem, moving plates are done recursively. We suppose that $n-1$ -plate problem is done then we solve n -plate problem

Divide and Conquer Algorithms

A **divide and conquer algorithm**:

- ▶ given a problem to be solved, split this into several smaller sub-problems.
- ▶ solve each of them recursively and then combine the sub-problem solutions so as to product a solution for the original problem.

Divide and Conquer Algorithms

A **divide and conquer algorithm**:

- ▶ given a problem to be solved, split this into several smaller sub-problems.
- ▶ solve each of them recursively and then combine the sub-problem solutions so as to product a solution for the original problem.

Traditionally, an algorithm is “divide and conquer” if it contains at least two recursive calls

Divide and Conquer Algorithms

Example: Fibonacci numbers 0, 1, 1, 2, 3, 5, 8, ...

```
1  int fibo(n){
2      if ((n == 0) || (n == 1))
3          return n;
4      return fibo(n-1) + fibo(n-2);
5  }
```

Divide and Conquer Algorithms

- ▶ Quicksort:

- ▶ Partition the array into two parts (smaller numbers in one part, larger numbers in the other part)
- ▶ Quicksort each of the parts

- ▶ Mergesort:

- ▶ Cut the array in half, and mergesort each half
- ▶ Combine the two sorted arrays into a single sorted array by merging them

Dynamic Programming

A **dynamic programming algorithm** remembers past results and uses them to find new results:

- ▶ cut the main problem into a set of simpler sub-problems
- ▶ solve each of those sub-problems just once, and store their solutions which can be used later.

This differs from Divide and Conquer, where sub-problems generally need not overlap.

Dynamic Programming

Example: Fibonacci numbers 0, 1, 1, 2, 3, 5, 8, ...

```
1  int fibo(n){
2      if ((n == 0) || (n == 1))
3          return n;
4      if fibo(n) != 0
5          return fibo(n);
6      return fibo(n-1) + fibo(n-2);
7  }
```

Since finding the i^{th} Fibonacci number involves finding all smaller Fibonacci numbers which are already calculated, the second recursive call has little work to do.

Greedy Algorithm

A **greedy algorithm** is an algorithm that follows the problem solving heuristic:

- ▶ take the best that we can get right now, without regard for future consequences.
- ▶ choose a local optimum at each step with the hope of finding a global optimum.

Greedy algorithms sometimes work well for optimization problems.

Example: Suppose you want to count out a certain amount of money, using the fewest possible bills and coins. A greedy algorithm would do this would be:

- ▶ At each step, take the largest possible bill or coin that does not overshoot.
- ▶ To make 151,000 VND, how many bills as few as possible? which should be the best solution?:

Example: Suppose you want to count out a certain amount of money, using the fewest possible bills and coins. A greedy algorithm would do this would be:

- ▶ At each step, take the largest possible bill or coin that does not overshoot.
- ▶ To make 151,000 VND, how many bills as few as possible? which should be the best solution?:
 - ▶ a 100,000 VND bill
 - ▶ a 50,000 VND bill
 - ▶ a 1,000 VND bill

Example: Suppose that in a certain currency system, we have 1 coin, 7 coins and 10 coins pieces.

Example: Suppose that in a certain currency system, we have 1 coin, 7 coins and 10 coins pieces.

- ▶ Using a greedy algorithm to count out 15 coins, you would get:
 - ▶ A 10 coins piece
 - ▶ Five 1 coin pieces, for a total of 15 coins
 - ▶ This requires **six coins**
- ▶ A better solution would be to use:
 - ▶ Two 7 coins pieces and one 1 coin piece
 - ▶ This only requires **three coins**

Backtracking Algorithm

A **backtracking algorithm** bases on recursion:

- ▶ starting with one possible move out of many available moves
- ▶ find next move from the starting point
- ▶ if this satisfies **given constraints**, continue next moves; else return to previous move

Sometimes, backtracking algorithms don't have solutions due to the constraints.

Backtracking Algorithm

Randomized Algorithms

A **randomized algorithm** is an algorithm that employs a degree of randomness as part of its logic.

- ▶ In Quicksort, using a random number to choose a pivot
- ▶ In Machine Learning, some techniques are “randomized” such as K-means, Self-organized Map, Random Forest, etc.

Complexity

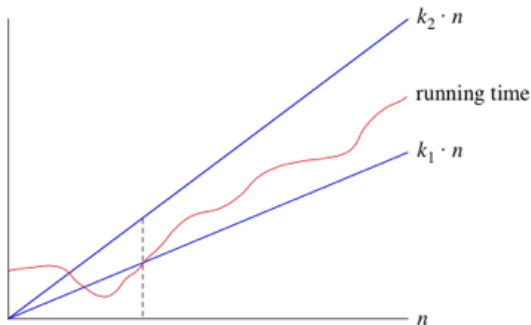
A theoretical evaluation to measure how good an algorithm is in term of running time and computational memory.

Assume that the running time of an algorithm is $T(n)$ with n objects.

- ▶ Big Θ : $T(n) = \Theta(f(n))$ if $\exists k_1, k_2, n_0 \in \mathbb{N}^+, \forall n \geq n_0$:
 $k_1 f(n) \leq T(n) \leq k_2 f(n)$
- ▶ Big O : $T(n) = O(f(n))$ if $\exists k, n_0 \in \mathbb{N}^+, \forall n \geq n_0$:
 $T(n) \leq k f(n)$
- ▶ Big Ω : $T(n) = \Omega(f(n))$ if $\exists k, n_0 \in \mathbb{N}^+, \forall n \geq n_0$:
 $T(n) \geq k f(n)$

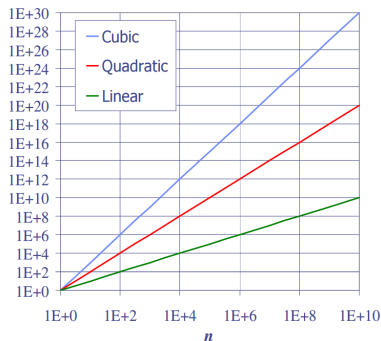
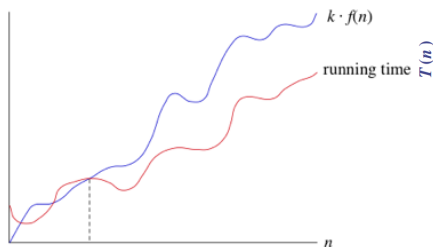
Algorithm complexity

Big Θ notation, an asymptotically tight bound on the running time, $T(n) = \Theta(f(n))$ if $\exists k_1, k_2, n_0 \in \mathbb{N}^+, \forall n \geq n_0$:

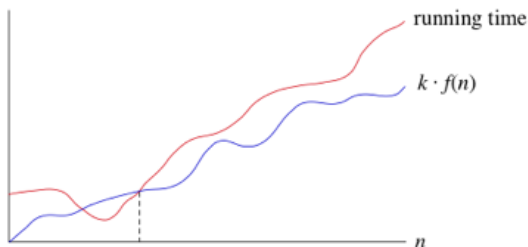
$$k_1 f(n) \leq T(n) \leq k_2 f(n)$$


Algorithm complexity

Big O notation, the asymptotic upper bounds of a running time,
 $T(n) = O(f(n))$ if $\exists k, n_0 \in \mathbb{N}^+, \forall n \geq n_0: T(n) \leq kf(n)$



Big Ω notation, the asymptotic lower bounds of a running time,
 $T(n) = \Omega(f(n))$ if $\exists k, n_0 \in \mathbb{N}^+, \forall n \geq n_0: T(n) \geq kf(n)$



Step 1 if $Max = a_1$ and $i = 2$, | 1 operation $\rightarrow \mathbf{O(1)}$

Algorithm complexity

Step 1 if $Max = a_1$ and $i = 2$,
Step 2 if $i > n$ then go to Step 6
Step 3 If $a_i > a_1$ then $Max = a_i$
Step 4 Increment i
Step 5 Go to Step 2

1 operation $\rightarrow \mathbf{O(1)}$

$n-1$ operations $\rightarrow \mathbf{O(n)}$

Algorithm complexity

Step 1	if $Max = a_1$ and $i = 2$,	1 operation $\rightarrow \mathbf{O(1)}$
Step 2	if $i > n$ then go to Step 6	
Step 3	If $a_i > a_1$ then $Max = a_i$	n-1 operations $\rightarrow \mathbf{O(n)}$
Step 4	Increment i	
Step 5	Go to Step 2	
Step 6	Return Max	1 operation $\rightarrow \mathbf{O(1)}$

Algorithm complexity

Step 1	if $Max = a_1$ and $i = 2$,	1 operation $\rightarrow \mathbf{O(1)}$
Step 2	if $i > n$ then go to Step 6	
Step 3	If $a_i > a_1$ then $Max = a_i$	n-1 operations $\rightarrow \mathbf{O(n)}$
Step 4	Increment i	
Step 5	Go to Step 2	
Step 6	Return Max	1 operation $\rightarrow \mathbf{O(1)}$

Complexity: $O(n + 1 + 1) = O(n)$

Some complexity examples:

- ▶ Any operation, statement, instruction: $S_1, \dots, S_k \rightarrow \mathbf{O(1)}$

Some complexity examples:

- ▶ Any operation, statement, instruction: $S_1, \dots, S_k \rightarrow \mathbf{O(1)}$
- ▶ $\text{for}\{i = 1; i \leq n; i++\}$
 $\{S_i\} \rightarrow \mathbf{O(n)}$

Some complexity examples:

- ▶ Any operation, statement, instruction: $S_1, \dots, S_k \rightarrow \mathbf{O(1)}$
- ▶ $\text{for}\{i = 1; i \leq n; i++\}$
 $\{S_i\} \rightarrow \mathbf{O(n)}$
- ▶ $\text{for}\{i = 1; i \leq n; i++\}\{$
 $\text{for}\{j = 1; j \leq n; j++\}$
 $\{S_i\} \rightarrow \mathbf{O(n^2)}$

Some complexity examples:

- ▶ Any operation, statement, instruction: $S_1, \dots, S_k \rightarrow \mathbf{O(1)}$
- ▶ $\text{for}\{i = 1; i \leq n; i++\}$
 $\{S_i\} \rightarrow \mathbf{O(n)}$
- ▶ $\text{for}\{i = 1; i \leq n; i++\}\{$
 $\text{for}\{j = 1; j \leq n; j++\}$
 $\{S_i\} \rightarrow \mathbf{O(n^2)}$

Several properties for complexity computation:

- ▶ $f(n) = O(h(n)) \rightarrow nf(n) = O(nh(n))$
- ▶ $f(n) = O(h(n)) \rightarrow kf(n) = O(h(n))$ where k is a constant
- ▶ $f(n) = O(h(n)) \rightarrow g(n) = O(y(n)) \rightarrow f(n)g(n) = O(h(n)y(n))$
- ▶ $f(n) + g(n) = \max(O(f(n)), O(y(n)))$

Algorithm complexity

- ▶ $O(1)$: Accessing any single element in an array takes constant time as only one operation has to be performed to locate it; or any arithmetic operations between two numbers, only one operation has to be done.
- ▶ $O(\ln(n))$: Algorithms taking logarithmic time are commonly found in operations on binary trees or when using binary search.
- ▶ $O(n)$: This linear complexity means that for large enough input sizes the running time increases linearly with the size of the input.
- ▶ $O(n^2)$: This quadratic complexity can be seen in algorithms for example bubble sort, insertion sort consisting of nested loops (a loop inside another loop) with the size of n .
- ▶ $O(n^3)$: This running time often requires from the multiplication of two $n \times n$ matrices
- ▶ $O(2^n)$: An exponential running time is used to found in the travelling salesman problem.

Algorithm complexity

$\log n$	\sqrt{n}	n	$n \log n$	$n(\log n)^2$	n^2
3	3	10	33	110	100
7	10	100	664	4,414	10,000
10	32	1,000	9,966	99,317	1,000,000
13	100	10,000	132,877	1,765,633	100,000,000
17	316	100,000	1,660,964	27,588,016	10,000,000,000

Example

```
1  for (int i = 1; i < n; i++)
2      for (int j = 1; j < n; j++)
3          a[i][j] = 0;
```

```
1  for (int i = 1; i < n; i++)
2      for (int j = 1; j < n; j++)
3          a[i][j] = 0;
4  for (int k = 1; k < n; k++)
5      a[k][k] = 1;
```

Example

```
1  int sum = 0;
2  int x, y;
3  for (int i = 1; i < n; i++)
4      for (int j = 1; j < n; j++){
5          for (int k = 1; k < 100; k++){
6              y = k;
7              x = 2*y;
8          }
9          sum = sum + i*j*k;
10 }
```