Introduction to Algorithms and Data Structures

Doan Nhat Quang

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

September 4, 2024

Objectives

Course objectives:

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

Objectives

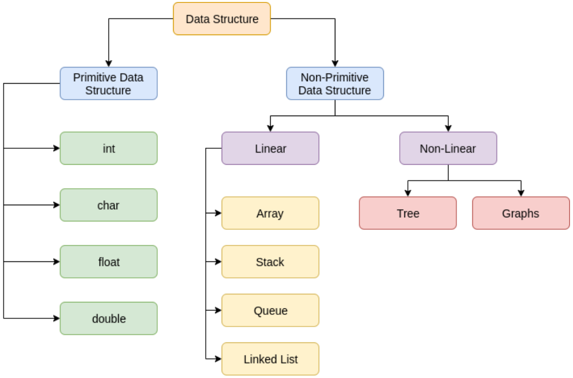
Why study Algorithms?

- 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

Objectives

Why study Data structure?

- Data structure can make the algorithms much simpler, easier to maintain, and often faster



Objectives

Basic algorithms

- How to search a number or a string of characters in the Google Search engine?
  $\rightarrow$ Solution: searching algorithms, indexing or sorting algorithms
- How to optimize the service algorithm for one or more elevator(s)?
  $\rightarrow$ Solution: scheduling algorithms, optimization algorithms
- How to maintain the products in store?
  $\rightarrow$ Solution: algorithms using the data structure: queue

Objectives

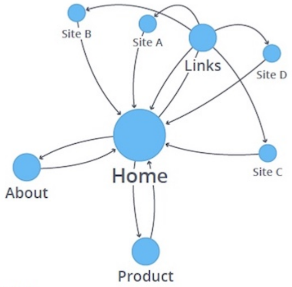
Advanced algorithms

- How to find people with the same interest?
- How to search an image or a video in Google Search engine?
- The traveling 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?
- and more

Real-world Applications

PageRank

- PageRank (PR) is an algorithm used by Google Search to rank websites in their search engine results. PageRank is a way of measuring the importance of website pages.
- How do you search a website in multiple servers?
- Data structures: Graph



Visualization of PageRank

Real-world Applications

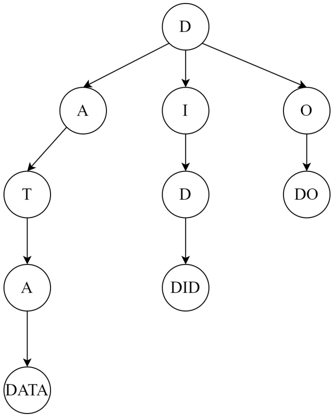
Travelling paths

- Finding the shortest paths is always practiced for **real-time** use.
- System: www.maps.google.com / www.ratp.fr
- Data structure: List of available vehicles, road network (graph)

Real-world Applications

Spelling and Auto-complete

- Data structure: Trie (a tree structure)
- Dictionnary: words are represented in a tree structure



Real-world Applications

Data Visualization

- Algorithms for visualization
- D3JS, GraphViz, PowerBI, etc.

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.

How do we use data structures with programming langangues?

- Algorithms must be implemented with **one programming language**.
- How many programming languages?
  $\rightarrow$ 500-700 programming languages excluding HTML, SQL, XML.
- Programming languages have syntax, libraries, functions, variables (more general data structures).
- Not all programming languages provide (pre-defined) data structures.
  $\rightarrow$ Users need to define their own data structures.
- Many data structures exist in litterature, we need to become proficient in understanding common data structures.

How do we use data structures with programming langangues?

User-defined data structures:

- More secure and confidential.
- Can be flexible and reusable for other problems.

Do you have data structures in other languages?

YES, main data structures

- C++: Lists, Stacks, Queues, Tree, Heap, Hashing, Graph.
- Python: Lists (Stack, Queue), Tuples and Sequences, Sets, Dictionaries.

Many data structures are integrated in each programming language.

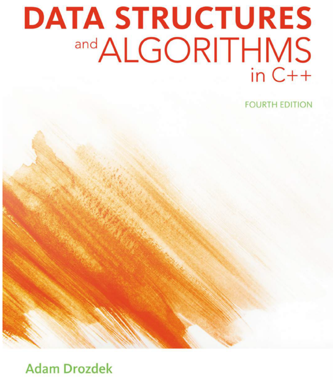
Why should we learn data structure using C?

- C is a basic programming language.
- C doesn't have any in-built data structures, except arrays.

Once we are familiar with data structures in C, we can implement them in any programming language.

YOU CANNOT USE BUILT-IN DATA STRUCTURES IN C/C++ FOR EXERCISE AND EXAM.

Textbooks



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

Development Tools

- 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

Example

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

Example

Suppose that a sequence of a_1, a_2, ... 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, ... 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, ... 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, ... 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, ... 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

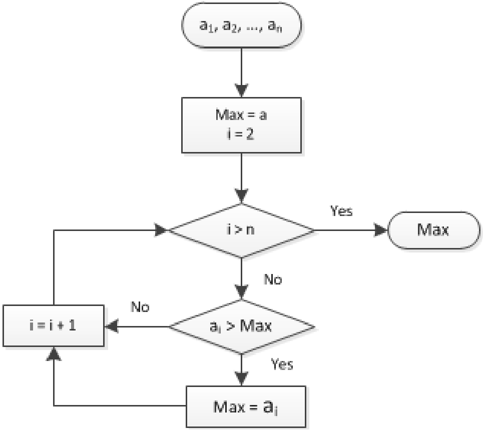
Suppose that a sequence of a_1, a_2, ... a_n $(n \geq 2)$ is available, find the maximum?

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

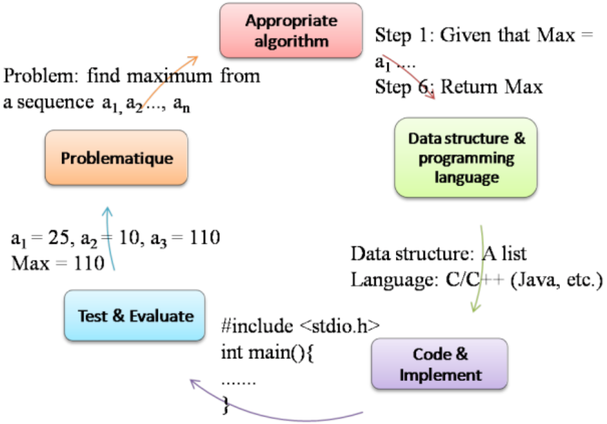
Suppose that a sequence of a_1, a_2, ... a_n ($n \geq 2$) is available, find the maximum?

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

Strategy to solve a problem:



Appropriate algorithm
Problem: find maximum from a sequence $a_1, a_2, \ldots, a_n$
Step 1: Given that Max = a_1 ….
Step 6: Return Max
Problematique
Data structure & programming language
$a_1 = 25, a_2 = 10, a_3 = 110$
Max = 110
Data structure: A list
Language: C/C++ (Java, etc.)
Test & Evaluate
Code & Implement

```
#include <stdio.h>
int main(){
.......
}
```

Algorithms

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

Data structure applications

Data Structure

- List of items in the cart when you visit an online shop
- List of possible actions (undo/redo) in a word editor
- Bitmap (array 2D) to store image pixels
- Graph to represent a group of persons and their relationship (Graph Theory, Graph Mining)
- Tree to arrange and index data like web pages, images, etc.

Definition

Computer program

A computer program is **a collection of instructions** a computer can execute to perform a specific task.

Definition

Computer program

A computer program is **a collection of instructions** a computer can execute to perform a specific task.

Algorithm

An algorithm is a finite sequence of well-defined instructions to **solve a specific problem or to perform a computation**.

- 3 constructs of algorithm: Input $\rightarrow$ Process $\rightarrow$ Output.

Definition

Computer program

A computer program is **a collection of instructions** a computer can execute to perform a specific task.

Algorithm

An algorithm is a finite sequence of well-defined instructions to **solve a specific problem or to perform a computation**.

- 3 constructs of algorithm: Input $\rightarrow$ Process $\rightarrow$ Output.

Data structure

A data structure is a data organization, management, and storage format that enables efficient access and computation.

Definition

Algorithm vs Program

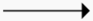
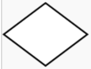
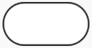
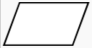
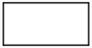
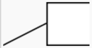
- A program can implement one or more algorithms;
- A program, there is always the idea that a computer will execute it while a person could execute an algorithm;
- A program is written in a programming language, while an algorithm is conceptual and can be described using language (including programming languages), flowcharts or pseudocode.

Program = Algorithm + Data structure.

Algorithm Representation

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

Algorithm Representation

Pseudo-code

- Pseudo-code is a high-level description
- Pseudo-code is concrete and easy for human comprehension
- Pseudo-code is usually 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
```

Algorithm design approaches

- **Top-down** approach emphasizes breaking down or dividing an algorithm into smaller modules (or functions). Each function is refined in more detail, probably in many additional levels, until they are no longer split.
- **Bottom-up** approach begins with the lowest-level functions of the algorithm, which are first specified in detail. These elements are then formed to create larger modules until a complete top-level system is formed.

Algorithms and approaches

A good algorithm must possess the following properties:

- Correctness: An algorithm may or may not have input. An algorithm has one or more outputs available when it terminates, which have a specific relation to the inputs. An algorithm must be correct if it takes the right input and produces the desired output.
- Finiteness: An algorithm must permanently terminate after a finite number of steps. Infinite loops should be avoided at all costs.

Algorithms and approaches

A good algorithm must possess the following properties:

- Definiteness: Each instruction must be precisely concrete and unambiguous. Each instruction must also be realized in a finite amount of time.
- Efficiency: An algorithm is always optimized to have low running time and low memory allocation as possible. A good algorithm should be concise and easy to understand and implement. Complexity, a theoretical measure, computes algorithmic complications. A well-optimized algorithm has low complexity.

Algorithm categories

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

Recursive Algorithms

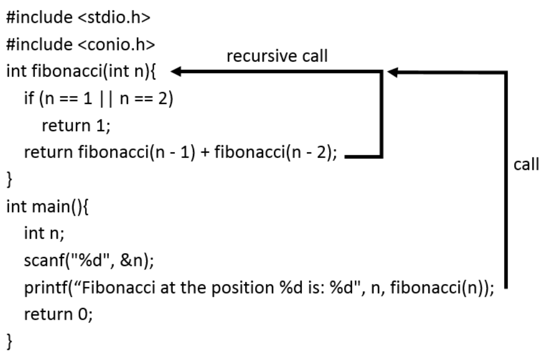
A simple recursive algorithm:

- 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



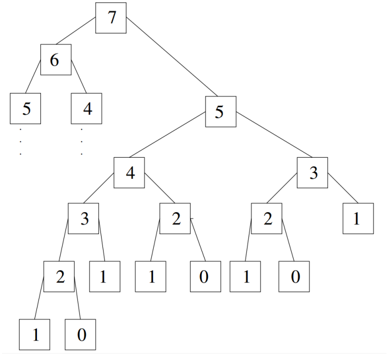
```
#include <stdio.h>
#include <conio.h>
int fibonacci(int n){
   if (n == 1 || n == 2)
      return 1;
   return fibonacci(n - 1) + fibonacci(n - 2);
}
int main(){
   int n;
   scanf("%d", &n);
   printf("Fibonacci at the position %d is: %d", n, fibonacci(n));
   return 0;
}
```

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

$\rightarrow$ Complexity is exponential, $O(2^n)$



Recursive Algorithms

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, ...

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

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 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, ...

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

Since finding the i^{th} Fibonacci number involves finding all smaller Fibonacci numbers 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 to find a global optimum.

Greedy algorithms sometimes work well for optimization problems.

Greedy Algorithm

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?:

Greedy Algorithm

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

Greedy Algorithm

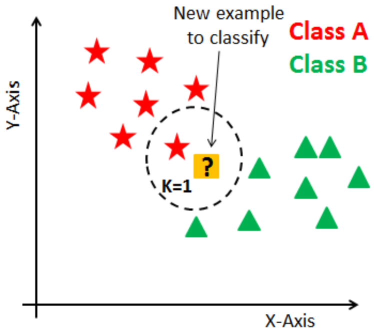
Example: Suppose that in a certain currency system, we have 1p, 7p and 10p pieces.

Greedy Algorithm

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

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

Greedy Algorithm



Label new data sample according to several data neighbors (k)

- Find k nearest neighbors of data sample
- Among these k data, if the number of data from any class is more common, the data sample is assigned to this class.

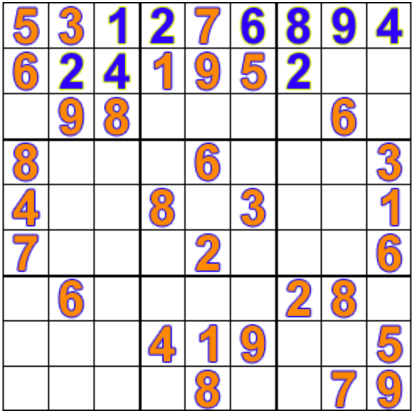
Backtracking Algorithm

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 constraints.

Backtracking Algorithm



5	3	1	2	7	6	8	9	4
6	2	4	1	9	5	2		
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

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.

Algorithm complexity

Complexity

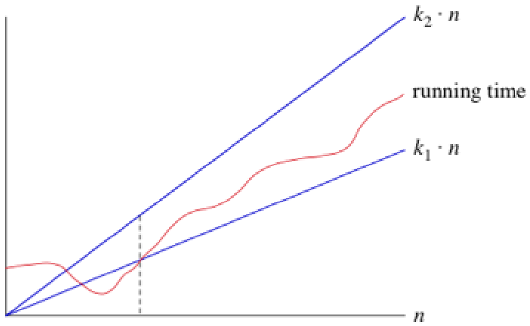
A theoretical evaluation measures how good an algorithm is in terms of running time and computational memory.

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

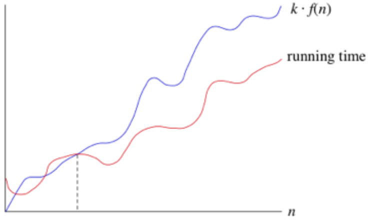
Algorithm complexity

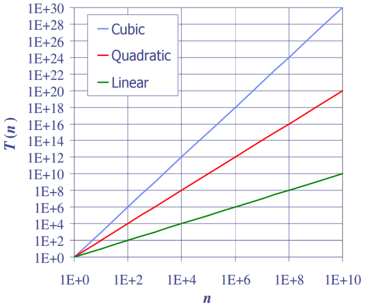
Big Θ notation, an asymptotically tight bound on the running time, $T(n) = \Theta(f(n))$ if $\exists k_1, k_2, n_0 \in \mathbb{N}^+, \forall n \geq n_0$: $k_1 f(n) \leq T(n) \leq k_2 f(n)$



Algorithm complexity

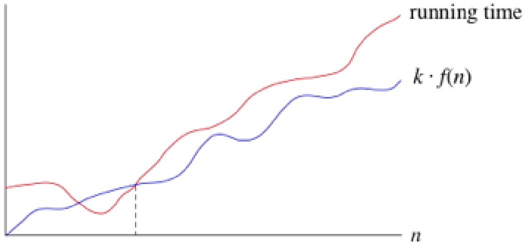
Big O notation, the asymptotic upper bounds of a running time, $T(n) = O(f(n))$ if $\exists k, n_0 \in \mathbb{N}^+, \forall n \geq n_0\text{:}\ \ T(n) \leq kf(n)$





Algorithm complexity

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



Algorithm complexity

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

Algorithm complexity

Step 1	if $Max = a_1$ and $i = 2$,	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	

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

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

Algorithm complexity

Some examples:

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

Algorithm complexity

Some examples:

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

Algorithm complexity

Some examples:

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

Algorithm complexity

Some examples:

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

Algorithm complexity

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 arithmeric 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 and insertion sort consisting of nested loops (a loop inside another loop) with the size of n.
- $O(n^3)$: This running time often requires the multiplication of two nxn matrices
- $O(2^n)$: An exponential running time is used to found in the traveling salesman problem.

Algorithm complexity

log n	$\sqrt{n}$	n	$nlogn$	$n(logn)^2$	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

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

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

Example

```
int sum = 0;
int x, y;
for (int i = 1; i <n; i++)
    for (int j = 1; j <n; j++){
        for (int k = 1; k < 100; k++){
            y = k;
            x = 2*y;
        }
        sum = sum + i*j*k;
    }
```