Course introduction, Recursion and Complexity of Algorithms

Data Structures and Algorithms Ngày 5 tháng 1 năm 2022

Dept. Computer Science

Faculty of Computer Science and Engineering

Ho Chi Minh University of Technology, VNU-HCM

Overview

1 DSA: Basic concepts

Some concepts on data
Algorithm
Pseudocode

2 Recursion

Recursion and the basic components of recursive algorithms
Properties of recursion
Designing recursive algorithms
Recursion and backtracking
Recursion implementation in C/C++

3 Complexity of Algorithms

Algorithm Efficiency
Asymptotic Analysis
Problems and common complexities
P and NP Problems

Sources of Materials

1. We would like to thank **Dr. The-Nhan LUONG**, a former instructor of our Department, for the composing of this document.
2. This document also uses figure, sentences and demo source code from the following sources:
   - The old presentation for course *Data Structures and Algorithms* edited by other members in our Department
   - Book entitled **Data Structures - A Pseudocode Approach with C++ (first edition, 2001)** written by Richard F. Gilberg and Behrouz A. Forouzan

Course learning outcomes

L.O.1	Determine the complexity of simple algorithms (polynomial time - nested loop - no recursive)
L.O.1.1	Give definition of Big-O notation
L.O.1.2	Determine complexity of simple polynomial algorithms
L.O.2	Manipulate basic data structures such as list, tree and graph
L.O.2.1	Describe and present basic data structures such as: array, linked list, stack, queue, tree, and graph
L.O.2.2	Implement basic methods for each of basic data structures: array, linked list, stack, queue, tree, and graph
L.O.3	Implement basic sorting and searching algorithms
L.O.3.1	Illustrate how searching algorithms work on data structures: array, linked list, stack, queue, tree, and graph
L.O.3.2	Illustrate how sorting algorithms work on an array
L.O.3.3	Implement necessary methods and proposed algorithms on a given data structure for problem solving

Basic concepts

What is Data?

(Source:

What is Data?

Data

Data is information that has been translated into a form that is more convenient to calculate, analyze.

Example

- Numbers, words, measurements, observations or descriptions of things.

- Qualitative data: descriptive information,
- Quantitative data: numerical information (numbers).
    - Discrete data can only take certain values (like whole numbers)
    - Continuous data can take any value (within a range)

Data type

Class of data objects that have the same properties.

Data type

1. A set of values
2. A set of operations on values

Example

Type	Values	Operations
integer	$-\infty, ..., -2, -1, 0, 1, 2, ..., \infty$	$*, +, -, \%, /, ++, --, ...$
floating point	$-\infty, ..., 0.0, ..., \infty$	$*, +, -, /, ...$
character	$\backslash 0$, ..., 'A', 'B', ..., 'a', 'b', ..., $\sim$	$<, >, ...$

Data structure

What is a data structure?

1. A combination of elements in which each is either a data type or another data structure
2. A set of associations or relationships (structure) that holds the data together

Example

An array is a number of elements of the same type in a specific order.

1	2	3	5	8	13	21	34

Abstract data type

The concept of abstraction:

- Users know what a data type can do.
- How it is done is hidden.

Definition

An **abstract data type** is a data declaration packaged together with the operations that are meaningful for the data type.

1. Declaration of data
2. Declaration of operations
3. Encapsulation of data and operations

Abstract data type

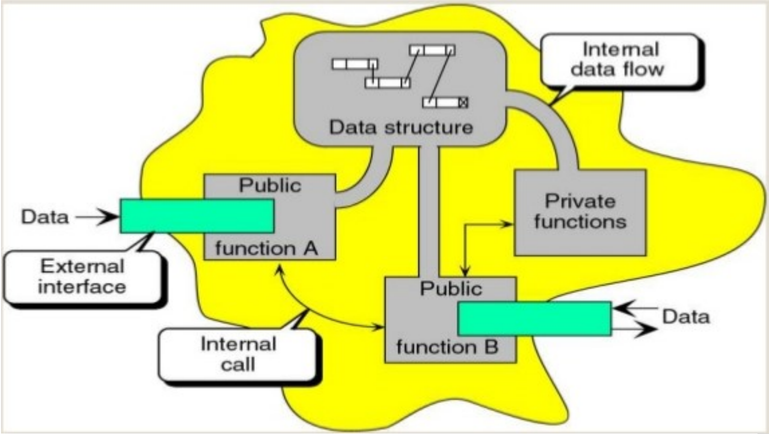


Figure: Abstract data type model (source: Slideshare)

Example: List

Interface

- **Data**: sequence of elements of a particular data type
- **Operations**: accessing, insertion, deletion

Implementation

- Array
- Linked list

Algorithm

What is an algorithm?

The logical steps to solve a problem.

What is a program?

Program = Data structures + Algorithms
(Niklaus Wirth)

Pseudocode

- The most common tool to define algorithms
- English-like representation of the algorithm logic
- Pseudocode = **English** + **code**

relaxed syntax being easy to read

instructions using basic control structures (sequential, conditional, iterative)

Pseudocode

Algorithm Header

- Name
- Parameters and their types
- Purpose: what the algorithm does
- Precondition: precursor requirements for the parameters
- Postcondition: taken action and status of the parameters
- Return condition: returned value

Algorithm Body

- Statements
- Statement numbers: decimal notation to express levels
- Variables: important data
- Algorithm analysis: comments to explain salient points
- Statement constructs: sequence, selection, iteration

Pseudocode: Example

Algorithm average

Pre nothing

Post the average of the input numbers is printed

```
i = 0
sum = 0
while all numbers not read do
    i = i + 1
    read number
    sum = sum + number
end
average = sum / i
print average
End average
```

Algorithm 1: How to calculate the average

Recursion and the basic components of recursive algorithms

Recursion

Definition

Recursion is a repetitive process in which an algorithm calls itself.

- Direct : A → A
- Indirect : A → B → A

Example

Factorial

$$Factorial(n) = \left[\begin{array}{ll} 1 & \text{if } n = 0 \\ n \times (n-1) \times ... \times 2 \times 1 & \text{if } n > 0 \end{array} \right.$$

Using recursion:

$$Factorial(n) = \left[\begin{array}{ll} 1 & \text{if } n = 0 \\ n \times Factorial(n-1) & \text{if } n > 0 \end{array} \right.$$

Basic components of recursive algorithms

Two main components of a Recursive Algorithm

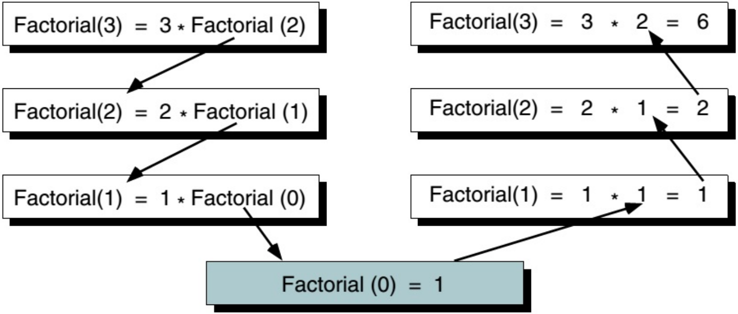
1. Base case (i.e. stopping case)
2. General case (i.e. recursive case)

Example

Factorial

$$Factorial(n) = \left[\begin{array}{lll} 1 & \text{if } n = 0 & \text{base} \\ n \times Factorial(n-1) & \text{if } n > 0 & \text{general} \end{array} \right.$$

Recursion



Hình: Factorial (3) Recursively
(Source: Data Structure - A pseudocode Approach with C++

Recursion

Factorial: Iterative Solution

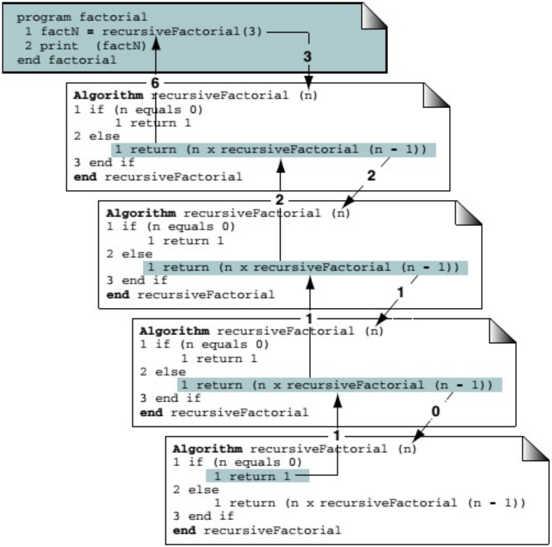
```
Algorithm iterativeFactorial(n)
Calculates the factorial of a number using a loop.
Pre: n is the number to be raised factorially
Post: n! is returned - result in factoN

i = 1
factoN = 1
while i <= n do
    factoN = factoN * i
    i = i + 1
end
return factoN
End iterativeFactorial
```

Recursion

Factorial: Recursive Solution

```
Algorithm recursiveFactorial(n)
Calculates the factorial of a number using a recursion.
Pre: n is the number to be raised factorially
Post: n! is returned

if n = 0 then
    return 1
else
    return n * recursiveFactorial(n-1)
end
End recursiveFactorial
```

Recursion



labelformat=empty

Hình: Calling a Recursive Algorithm (source: Data Structure - A pseudocode Approach with C++)

Properties of recursion

Properties of all recursive algorithms

- A recursive algorithm solves the large problem by using its solution to a simpler sub-problem

- Eventually the sub-problem is simple enough that it can be solved without applying the algorithm to it recursively.
  $\rightarrow$ This is called the base case.

Designing recursive algorithms

The Design Methodology

Every recursive call must either solve a part of the problem or reduce the size of the problem.

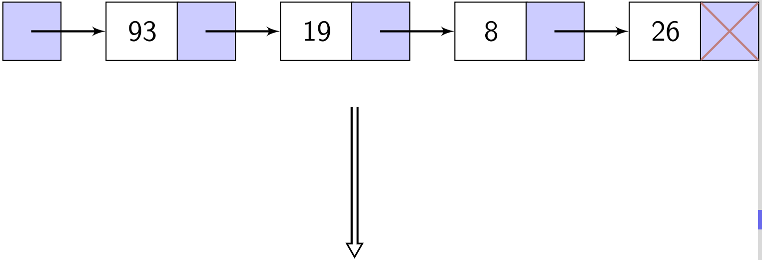
Rules for designing a recursive algorithm

1. Determine the base case (stopping case).
2. Then determine the general case (recursive case).
3. Combine the base case and the general cases into an algorithm.

Limitations of Recursion

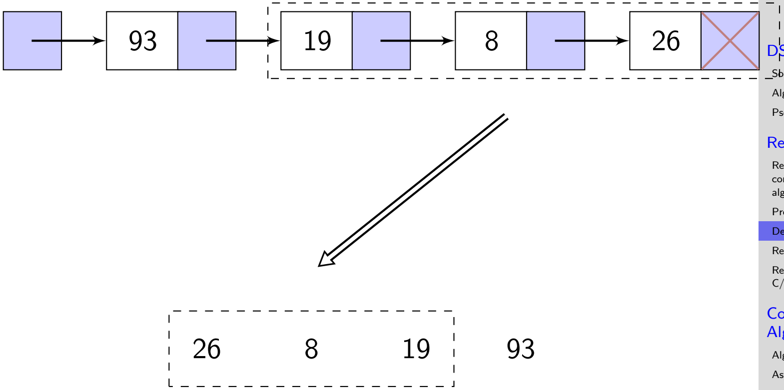
- A recursive algorithm generally runs more slowly than its nonrecursive implementation.
- BUT, the recursive solution shorter and more understandable.

Print List in Reverse



26 8 19 93

Print List in Reverse



Print List in Reverse

```
Algorithm printReverse(list)
Prints a linked list in reverse.
Pre: list has been built
Post: list printed in reverse

if list is null then
    return
end
printReverse (list -> next)
print (list -> data)
End printReverse
```

Greatest Common Divisor

Definition

$$\gcd(a, b) = \left[\begin{array}{ll} a & \text{if } b = 0 \\ b & \text{if } a = 0 \\ \gcd(b, a \mod b) & \text{otherwise} \end{array}\right.$$

Example

$\gcd(12, 18) = 6$
$\gcd(5, 20) = 5$

Greatest Common Divisor

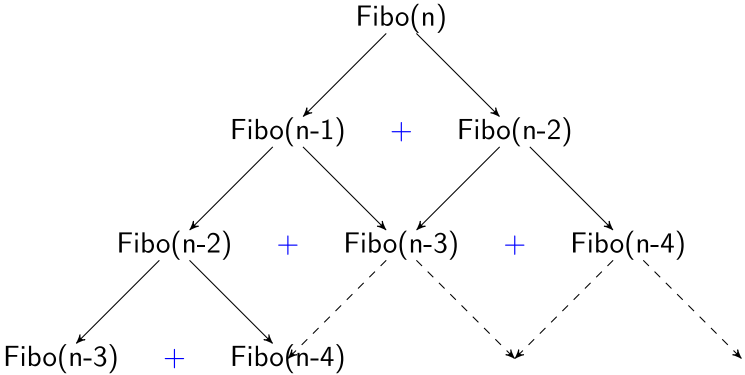
```
Algorithm gcd(a, b)
Calculates greatest common divisor using the Euclidean
  algorithm.
Pre: a and b are integers
Post: greatest common divisor returned

if b = 0 then
    return a
end
if a = 0 then
    return b
end
return gcd(b, a mod b)
End gcd
```

Fibonacci Numbers

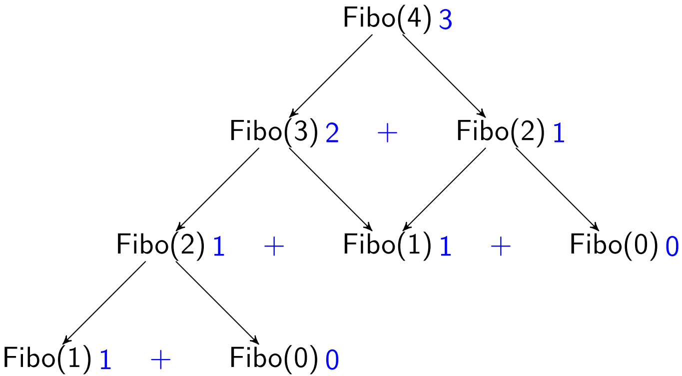
Definition

$$Fibo(n) = \left[\begin{array}{ll} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ Fibo(n-1) + Fibo(n-2) & \text{otherwise} \end{array} \right.$$

Fibonacci Numbers



Fibonacci Numbers



Result

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Fibonacci Numbers

```
Algorithm Fibo(n)
Calculates the n^th Fibonacci number.
Pre: n is postive integer
Post: the n^th Fibonnacci number returned

if n = 0 or n = 1 then
    return n
end
return Fibo(n-1) + Fibo(n-2)
End fib
```

Fibonacci Numbers

No	Calls	Time	No	Calls	Time
1	1	$<$ 1 sec.	11	287	$<$ 1 sec.
2	3	$<$ 1 sec.	12	465	$<$ 1 sec.
3	5	$<$ 1 sec.	13	753	$<$ 1 sec.
4	9	$<$ 1 sec.	14	1,219	$<$ 1 sec.
5	15	$<$ 1 sec.	15	1,973	$<$ 1 sec.
6	25	$<$ 1 sec.	20	21,891	$<$ 1 sec.
7	41	$<$ 1 sec.	25	242,785	1 sec.
8	67	$<$ 1 sec.	30	2,692,573	7 sec.
9	109	$<$ 1 sec.	35	29,860,703	1 min.
10	177	$<$ 1 sec.	40	331,160,281	13 min.

The Towers of Hanoi

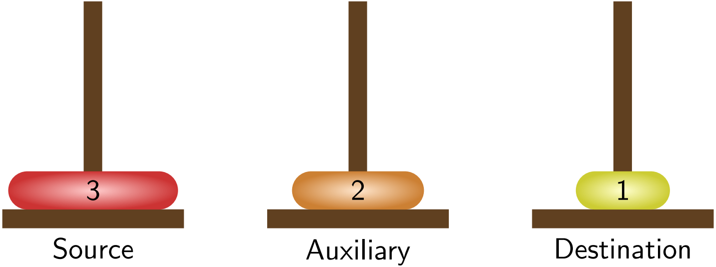
Move disks from Source to Destination using Auxiliary:

1. Only one disk could be moved at a time.
2. A larger disk must never be stacked above a smaller one.
3. Only one auxiliary needle could be used for the intermediate storage of disks.

The Towers of Hanoi

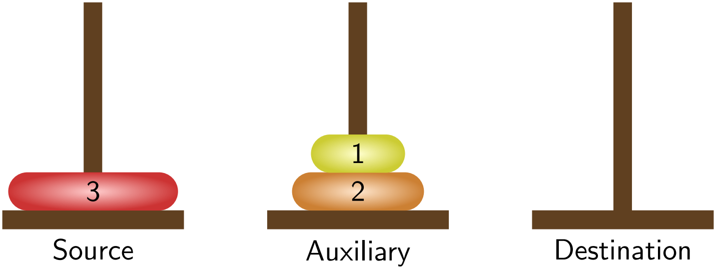
Moved disc from pole 1 to pole 3.

The Towers of Hanoi



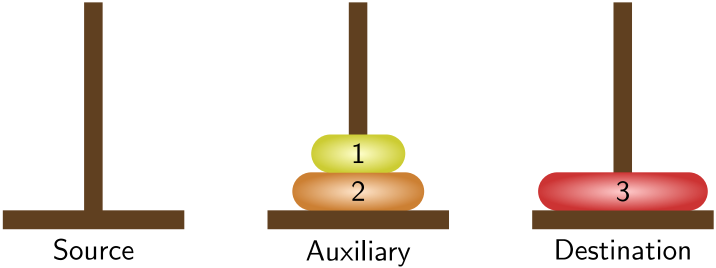
Moved disc from pole 1 to pole 2.

The Towers of Hanoi



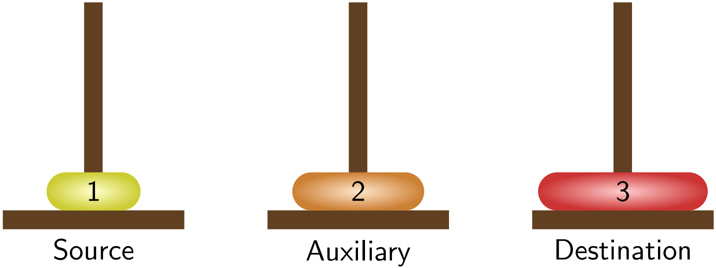
Moved disc from pole 3 to pole 2.

The Towers of Hanoi



Moved disc from pole 1 to pole 3.

The Towers of Hanoi



Moved disc from pole 2 to pole 1.

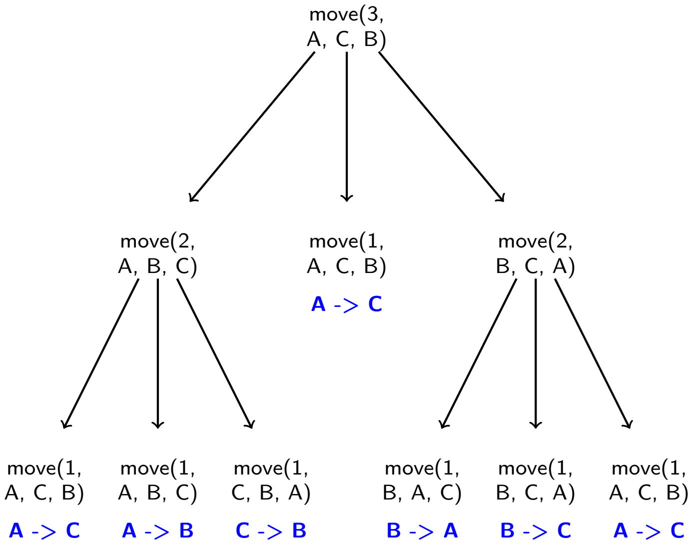
The Towers of Hanoi

Moved disc from pole 2 to pole 3.

The Towers of Hanoi

Moved disc from pole 1 to pole 3.

The Towers of Hanoi



The Towers of Hanoi : General

move(n, A, C, B)

move(n-1, A, B, C)   move(1, A, C, B)   move(n-1, B, C, A)

Complexity

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

The Towers of Hanoi

Complexity

$$T(n) = 1 + 2T(n-1)$$
$$=> T(n) = 1 + 2 + 2^2 + ... + 2^{n-1}$$
$$=> T(n) = 2^n - 1$$
$$=> T(n) = O(2^n)$$

- With 64 disks, total number of moves: $2^{64} - 1 \approx 2^4 \times 2^{60} \approx 2^4 \times 10^{18} = 1.6 \times 10^{19}$
- If one move takes 1s, 2^{64} moves take about 5×10^{11} years (500 billions years).

The Towers of Hanoi

```
Algorithm move(val disks <integer>, val source
  <character>, val destination <character>, val
  auxiliary <character>)
Move disks from source to destination.
Pre: disks is the number of disks to be moved
Post: steps for moves printed
print("Towers: ", disks, source, destination, auxiliary)
if disks = 1 then
|   print ("Move from", source, "to", destination)
else
|   move(disks - 1, source, auxiliary, destination)
|   move(1, source, destination, auxiliary)
|   move(disks - 1, auxiliary, destination, source)
end
return
End move
```

Recursion and backtracking

Backtracking

Definition

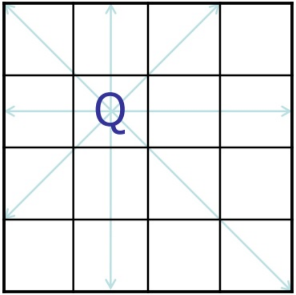
A process to go back to previous steps to try unexplored alternatives.

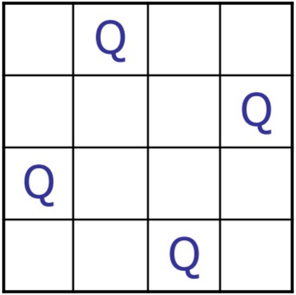
backtracking.jpg

Hình: Goal seeking

Eight Queens Problem

Place eight queens on the chess board in such a way that no queen can capture another.





Eight Queens Problem

Algorithm putQueen(ref board <array>, val r <integer>)
Place remaining queens safely from a row of a chess board.

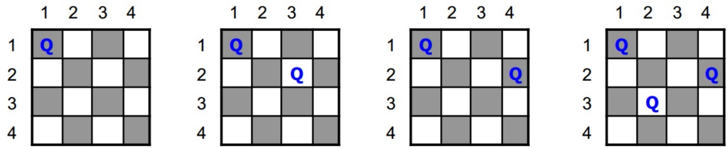
Pre: `board` is nxn array representing a chess board
`r` is the row to place queens onwards

Post: all the remaining queens are safely placed on the board; or backtracking to the previous rows is required

Eight Queens Problem

```
for every column c on the same row r do
    if cell r,c is safe then
        place the next queen in cell r,c
        if r < n-1 then
            putQueen (board, r + 1)
        else
            output successful placement
        end
        remove the queen from cell r,c
    end
end
return
End putQueen
```

Eight Queens Problem



Recursion implementation in C/C++

Fibonacci Numbers

```
#include <iostream>
using namespace std;

long fib(long num);

int main () {
  int num;
  cout << "What␣Fibonacci␣number
␣␣␣␣␣␣␣␣␣␣do␣you␣want␣to␣calculate?␣";
  cin >> num;
  cout << "The␣" << num << "th␣Fibonacci␣number
␣␣␣␣␣␣␣␣␣␣is:␣" << fib(num) << endl;
  return 0;
}

long fib(long num) {
  if (num == 0 || num == 1)
    return num;
  return fib(num-1) + fib(num-2);
}
```

The Towers of Hanoi

```
#include <iostream>
using namespace std;

void move(int n, char source,
          char destination, char auxiliary);

int main () {
  int numDisks;
  cout << "Please␣enter␣number␣of␣disks:␣";
  cin >> numDisks;
  cout << "Start␣Towers␣of␣Hanoi" << endl;
  move(numDisks, 'A', 'C', 'B');
  return 0;
}
```

The Towers of Hanoi

```
void move(int n, char source,
          char destination, char auxiliary){
  static int step = 0;

  if (n == 1)
    cout << "Step " << ++step << ": Move from "
      << source << " to " << destination << endl;
  else {
    move(n-1, source, auxiliary, destination);
    move(1, source, destination, auxiliary);
    move(n - 1, auxiliary, destination, source);
  }
  return;
}
```

Algorithm Efficiency

Compare Algorithms

- Given 2 or more algorithms to solve the same problem, how do we select the best one?
- Some criteria for selecting an algorithm
  - Is it easy to implement, understand, modify?
  - How long does it take to run it to completion?
  - How much of computer memory does it use?
- Software engineering is primarily concerned with the first criteria
- In this course we are interested in the second and third criteria

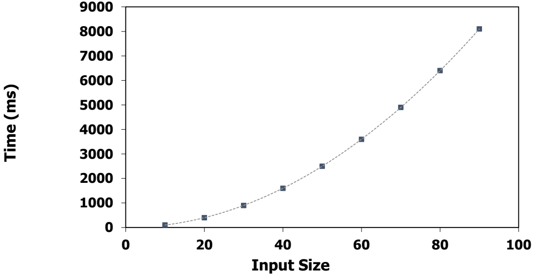
Algorithm Efficiency

- **Time complexity**: the amount of time that an algorithm needs to run to completion.
- **Space complexity**: the amount of memory an algorithm needs to run.
- We will occasionally look at space complexity, but we are mostly interested in time complexity in this course.
- Thus in this course the better algorithm is the one which runs faster (has smaller time complexity).

How to Calculate Running time

- Most algorithms transform input objects into output objects.

- The running time of an algorithm typically grows with the input size $f(n)$.



How to Calculate Running Time

- Even on inputs of the same size, running time can be very different.
  - Example: algorithm that searches an array containing n integers to find the one with a particular value K (assume that K appears exactly once in the array)
- Idea: analyze running time in the
  - best case
  - worst case
  - average case

How to Calculate Running Time

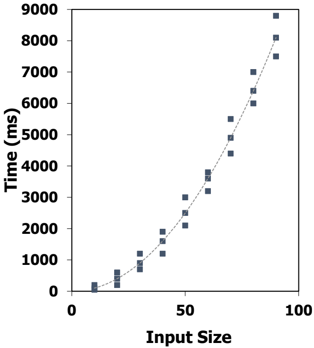
- Best case running time is usually useless
- Average case time is very useful but often difficult to determine
- We focus on the worst case running time
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics

Evaluations of Running Time

- There are two ways to compare running time between agorithms:
  - Experimental evaluation
  - Theoretical evaluation

Experimental Evaluation of Running Time

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Measure accurately the actual running time
- Plot the results



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment
- In order to compare two algorithms, the same hardware and software environments must be used

Theoretical Analysis of Running Time

- Uses a pseudo-code description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, **n**
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Primitive Operations

- For theoretical analysis, we will count **primitive** or **basic** operations, which are simple computations performed by an algorithm
- Example:
  - Evaluating an expression: $x^2 + 3x$
  - Assigning a value to a variable: $x = y$
  - Indexing into an array: $a[10]$
  - Calling a function: $mySwap(a, b)$
  - Returing from a function: $return(x)$

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

Algorithm findMax(a, n)

```
currentMax = a[0]            //2
i = 1                        //2
while (i < n)                //n
  if (currentMax < a[i])     //2n-2
    currentMax = a[i]        //2n-2
  i = i + 1                  //2n-2
return currentMax            //1
```

Estimating Running Time

- Algorithm findMax() executes $7n - 1$ primitive operations in the worst case. Define:
  - a = Time taken by the fastest primitive operation
  - b = Time taken by the slowest primitive operation
- Let f(n) be worst-case time of arrayMax. Then $a(7n - 1) \leq f(n) \leq b(7n - 1)$
- Hence, the running time f(n) is bounded by two linear functions

Growth Rate of Running Time

- Changing the hardware/software environment
  - Affects $f(n)$ by a constant factor, but
  - Does not alter the growth rate of $f(n)$
- Thus we focus on the big-picture which is the growth rate of an algorithm
- The linear growth rate of the running time $f(n)$ is an intrinsic property of algorithm findMax()

Linear Loops

```
for (i = 0; i < n; i++)
   application code
```

The number of times the body of the loop is replicated is n.

$$\mathbf{f(n) = n}$$

```
for (i = 0; i < n; i += 2)
   application code
```

The number of times the body of the loop is replicated is $n/2$.

$$\mathbf{f(n) = n/2}$$

Logarithmic Loops

Multiply loops

```
i = 1
while (i <= n)
   application code
   i = i x 2
```

Divide loops

```
i = n
while (i >= 1)
   application code
   i = i / 2
```

The number of times the body of the loop is replicated is

$$\mathbf{f(n)} = \log_2 n$$

Nested Loops

Iterations = Outer loop iterations × Inner loop iterations

Example

```
i = 1
while (i <= n)
   j = 1
   while (j <= n)
      application code
      j = j * 2
   i = i + 1
```

The number of times the body of the loop is replicated is

$$f(n) = n \log_2 n$$

Quadratic Loops

Example

```
i = 1
while (i <= n)
   j = 1
   while (j <= n)
      application code
      j = j + 1
   i = i + 1
```

The number of times the body of the loop is replicated is

$$f(n) = n^2$$

Dependent Quadratic Loops

Example

```
i = 1
while (i <= n)
   j = 1
   while (j <= i)
      application code
      j = j + 1
   i = i + 1
```

The number of times the body of the loop is replicated is

$$1 + 2 + \ldots + n = n(n + 1)/2$$

Constant Factors

- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^3n + 10^5$ is a linear function
  - $10n^2 + 10^4n$ is a quadratic function
- How do we get rid of the constant factors to focus on the essential part of the running time?

Asymptotic Analysis

- Algorithm efficiency is considered with only big problem sizes.
- We are not concerned with an exact measurement of an algorithm's efficiency.
- Terms that do not substantially change the function's magnitude are eliminated.

Asymptotic Analysis

Asymptotic Notations

There are three common notations for asymptotic analysis:

- Big-Oh: $O(.)$
- Big-Omega: $\Omega(.)$
- Big-Theta: $\Theta(.)$

Big-Oh notation

- The big-Oh notation is used widely to characterize running times and space bounds
- The big-Oh notation allows us to ignore constant factors and lower order terms and focus on the main components of a function which affect its growth

Big-Oh notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that: $f(n) \leq c.g(n)$ for $n \geq n_0$
- Example: $2n + 10$ is $O(n)$
- Why?

Big-Oh Rules

Example

$f(n) = c.n \Rightarrow f(n) = O(n)$
$f(n) = n(n+1)/2 = n^2/2 + n/2 \Rightarrow f(n) = O(n^2)$

- Set the coefficient of the term to one.
- Keep the largest term and discard the others.

Some example of Big-O:

$1 \quad \log_2 n \quad n \quad n\log_2 n \quad n^2 \ldots n^k \ldots 2^n \quad n!$

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

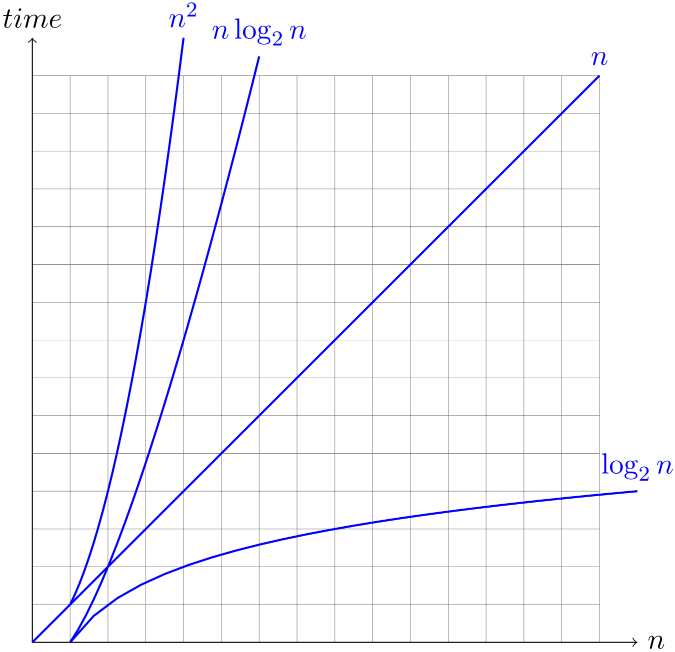
Standard Measures of Efficiency

Efficiency	Big-O	Iterations	Est. Time
logarithmic	$O(\log_2 n)$	14	microseconds
linear	$O(n)$	10 000	0.1 seconds
linear log	$O(n \log_2 n)$	140 000	2 seconds
quadratic	$O(n^2)$	10000^2	15-20 min.
polynomial	$O(n^k)$	10000^k	hours
exponential	$O(2^n)$	2^{10000}	intractable
factorial	$O(n!)$	$10000!$	intractable

Assume instruction speed of 1 microsecond and 10 instructions in loop.

$n = 10000$

Standard Measures of Efficiency



Relatives of Big-Oh

- big-Omega
  - $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c.g(n)$ for $n \geq n_0$
- big-Theta
  - $f(n)$ is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c'.g(n) \geq f(n) \geq c''.g(n)$ for $n \geq n_0$

Intuition for Asymptotic Notation

- Big-Oh: $f(n)$ is $O(g(n))$ if $f(n)$ is asymptotically less than or equal to $g(n)$
- Big-Omega: $f(n)$ is $\Omega(g(n))$ if $f(n)$ is asymptotically greater than or equal to $g(n)$
- Big-Theta: $f(n)$ is $\Theta(g(n))$ if $f(n)$ is asymptotically equal to $g(n)$

Problems and common complexities

Binary search

Recurrence Equation (Phương trình hồi quy)

An equation or inequality that describes a function in terms of its value on smaller input.

1	2	3	5	8	13	21	34	55	89

$$T(n) = 1 + T(n/2) \Rightarrow T(n) = O(\log_2 n)$$

Binary search

- Best case: when the number of steps is smallest. $T(n) = O(1)$
- Worst case: when the number of steps is largest. $T(n) = O(\log_2 n)$
- Average case: in between. $T(n) = O(\log_2 n)$

Sequential search

8	5	21	2	1	13	4	34	7	18

- Best case: $T(n) = O(1)$
- Worst case: $T(n) = O(n)$
- Average case: $T(n) = \sum_{i=1}^{n} i.p_i$
  p_i : probability for the target being at a[i]
  $p_i = 1/n \Rightarrow T(n) = (\sum_{i=1}^{n} i)/n = O(n(n+1)/2n) = O(n)$

Quick sort

19	8	3	15	28	10	22	4	12	83

Recurrence Equation

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

- Best case: $T(n) = O(n \log_2 n)$
- Worst case: $T(n) = O(n^2)$
- Average case: $T(n) = O(n \log_2 n)$

P and NP Problems

P and NP Problems

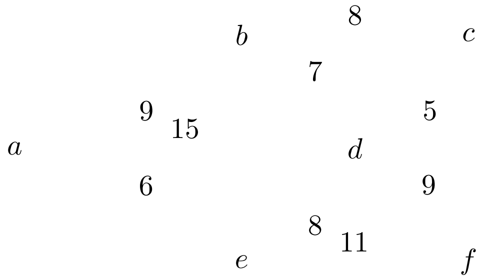
- P: Polynomial (can be solved in polynomial time on a deterministic machine).
- NP: Nondeterministic Polynomial (can be solved in polynomial time on a nondeterministic machine).

P and NP Problems

Travelling Salesman Problem:

A salesman has a list of cities, each of which he must visit exactly once. There are direct roads between each pair of cities on the list.

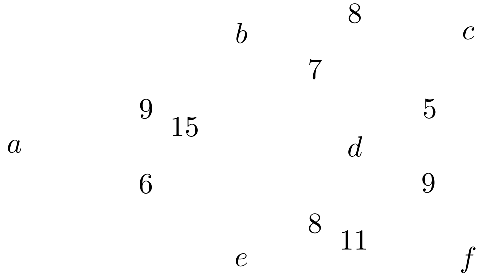
Find the route the salesman should follow for the shortest possible round trip that both starts and finishes at any one of the cities.



P and NP Problems

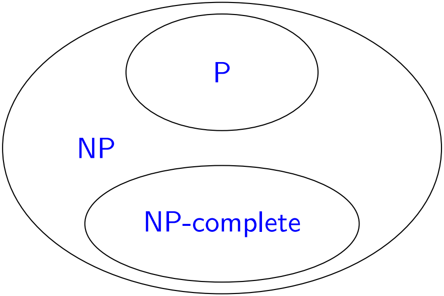
Travelling Salesman Problem:

Deterministic machine:
$f(n) = n(n-1)(n-2)\dots 1 = O(n!)$
$\Rightarrow$ NP problem



P and NP Problems

NP-complete: NP and every other problem in NP is polynomially reducible to it.



$$P = NP?$$

THANK YOU.