Algorithm Analysis tools

Dr.Rasha Elagamy

Algorithm Analysis: Motivation

- **A problem can be solved in many different ways**
  - Single problem, many algorithms
- **Which of the several algorithms should I choose?**
  - We use algorithm analysis to answer this question
    - o Writing a working program is **not good enough**
    - o The program may be inefficient!
    - o If the program runs on a large data set, then the running time becomes an issue

What is algorithm analysis?

- **A methodology to predict the resources that the algorithm requires**
  - Computer memory : the space it uses
  - Computational time: the time it takes to execute
- We'll focus on computational time
  - It does not mean memory is not important
  - Generally, there is **a trade-off** between the two factors
    - **Space-time trade-off** is a common term

How to analyse algorithms? (1)

❑ **Experimental Approach**

- Implement algorithms as programs and run them on computers

- Not a good approach, though!

  - Results only for a limited set of test inputs
  - Difficult comparisons due to the experiment environments (need the same computers, same operating systems, etc.)
  - Full implementation and execution of an algorithm

- **We need an approach which allows us to avoid experimental study**

Machine & programming language independence

- The programming language chosen to implement the algorithm
- The quality of the compiler
- The evaluation of efficiency should be as machine independent as possible.
  - count the number of basic operations the algorithm performs.
  - calculate how this number depends on the size of the input.

Machine & programming language independence

- The programming language chosen to implement the algorithm
- The quality of the compiler
- The evaluation of efficiency should be as machine independent as possible.
  - count the number of basic operations the algorithm performs.
  - calculate how this number depends on the size of the input.

A basic operation is an operation
which takes a constant amount of time to execute.

Example Basic Operations:

Addition, Subtraction, Multiplication, Memory Access..etc...

Non-basic Operations:

Sorting, Searching ....etc...

How to analyse algorithms? (2)

- Theoretical Approach
  - General methodology for analysing the running time
    - Considers all possible inputs
    - Evaluates algorithms in a way that is independent from the hardware and software environments
    - Analyses an algorithm without implementing it

How to analyse algorithms? (3)

❑ Theoretical Approach (cont.)

- Count only primitive operations used in an algorithm
- Associate each algorithm with a function ***f(n)*** that characterises the running time of the algorithm as a function of the input size *n*
  - A good approximation of the total number of primitive operations

Primitive Operations

- Basic computations performed by an algorithm
- Each operation corresponding to a low-level instruction with a constant execution time
- Largely independent from the programming language
- Examples
  - Evaluating an expression (`x + y`)
  - Assigning a value to a variable (`x ←5`) 1 operation
  - Comparing two numbers (`x < y`)
  - Indexing into an array (`A[i]`)
  - Calling a method (`mycalculator.sum()`)
  - Returning from a method (**`return`** `result`)

Primitive Operations

- Examples of primitive operations:
  - Evaluating an expression → x2+cy
  - Assigning a value to a variable → count ← count+1
  - Indexing into an array → Array[5]
  - Calling a method → mySort(myArray, n)
  - Returning from a method → return(count)

Low Level Algorithm Analysis

- Based on primitive operations (low-level computations independent from the programming language)

- E.g.:
  - Make an addition = 1 operation
  - Calling a method or returning from a method = 1 operation
  - Index in an array = 1 operation
  - Comparison = 1 operation etc.
- Method: Inspect the pseudo-code and count the number of primitive operations executed by the algorithm

Counting Primitive Operations (1)

- **Total number of primitive operations executed**
  - is the running time of an algorithms
  - is a function of the input size
- **Example**

Algorithm	# operations
`Algorithm ArrayMax(A, n)`	**# operations**
`currentMax ←A[0]`	**2**: (1 +1)
`for i←1;i<n; i←i+1 do`	**3n-1**: (1 + n+2(n- 1))
`    if A[i]>currentMax then`	**2(n − 1)**
`        currentMax ←A[i]`	**2(n − 1)** **(at most!)**
`    endif`	
`endfor`	
`return currentMax`	**1**
	Total: 7n − 2

Counting Primitive Operations (2)

- **Simpler approach!**
  - block or group of constant primitive operation can be combined!
- Example

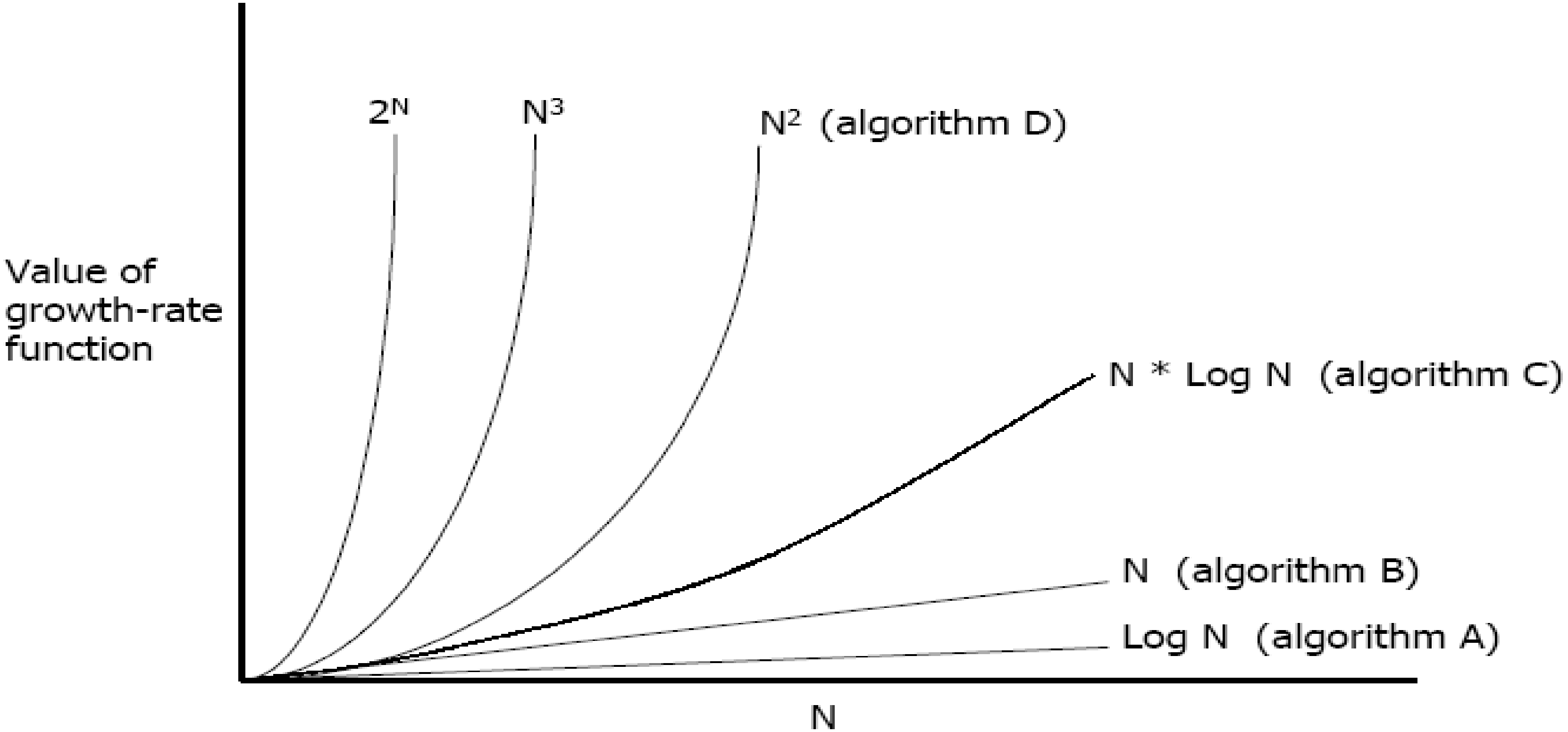
Algorithm	# operations
`Algorithm ArrayMax(A, n)`	# operations
`currentMax ←A[0]`	1
`for i←1;i<n; i←i+1 do`	n
`    if A[i]>currentMax then`	1(n-1)
`        currentMax ←A[i]`	1(n-1) **(at most!)**
`    endif`	
`endfor`	
`return currentMax`	1
	Total: 3n

Algorithm efficiency: growth rate

- An algorithm's time requirements can be expressed as a function of (problem) input size
- Problem size depends on the particular problem:
  - For a **search problem**, the problem size is the number of elements in the search space
  - For a **sorting problem**, the problem size is the number of elements in the given list
- How quickly **the time** of an algorithm **grows** as a function of problem size -- this is often called an **algorithm's growth rate**

Algorithm growth rate

Which algorithm is the most efficient? [The one with the growth rate Log N.]



Algorithmic time complexity

- Rather than counting the exact number of primitive operations, we approximate the runtime of an algorithm *as a function of data size* – time complexity
- The type of that function will depend on the problem (constant,linear,quadratic,….).
- Algorithms A, B, C and D (previous slide) belong to different complexity classes
- We’ll not cover complexity classes in detail – they will be covered in Algorithm Analysis course, in a later semester
- We’ll briefly discuss seven basic functions which are often used in complexity analysis

How to Calculate Running time

- Most algorithms transform input objects into output objects

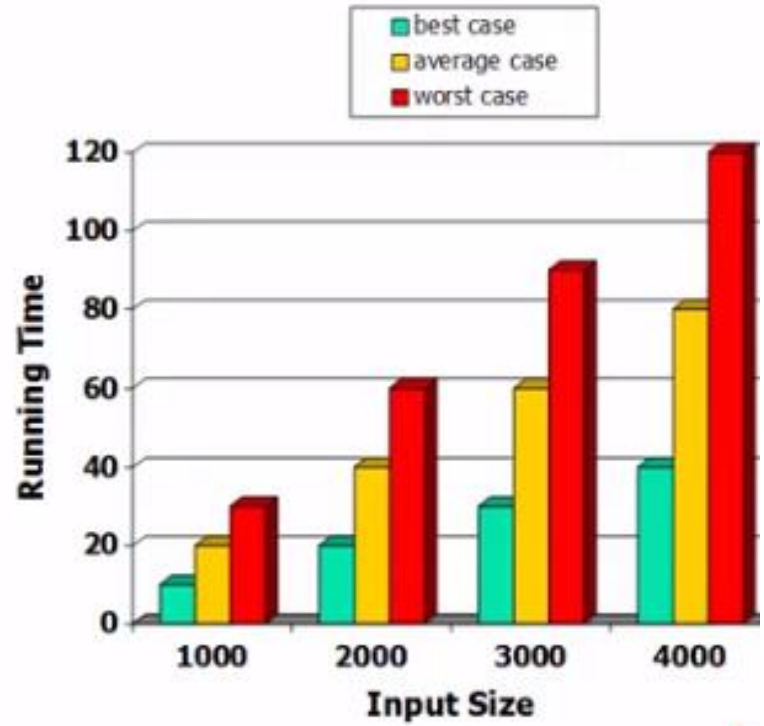
- The running time of an algorithm typically grows with the input size
  - Analyze running time as a function of input size

How to Calculate Running Time

- The running time of an algorithm varies with the inputs, and typically grows with the size of the inputs.
- 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



Seven basic function

1. Constant function $f(n) = c$
2. Linear function $f(n) = n$
3. Quadratic function $f(n) = n^2$
4. Cubic function $f(n) = n^3$
5. Log function $f(n) = \log n$
6. Log linear function $f(n) = n \log n$
7. Exponential function $f(n) = b^n$