Algorithms and Data Structures
Lecture notes: Definitions, terminology, Brassard Chap. 2

Lecturer: Michel Toulouse

Vietnamese-German University
michel.toulouse@vgu.edu.vn

4 avril 2016

Outline

- Definitions of terms like *algorithm*, *input size*, *formal parameters*, *output*, *running time*, *time complexity*, *space complexity*, *basic operations*, *rate of growth*, etc.
- Distinction between terms like *problem* and *problem's instance*, or *algorithm* and *program*.
- Introduction of the basic procedure and steps to analyze algorithms.
- For a YouTube history of computers, see `https://www.youtube.com/watch?v=hGXZjlDSXyg`

Algorithms

- What is an algorithm ?
- Origin of the word : al-Khowarizmi, Persian scholar (780-850), early ”textbook” on arithmetic operations.
- The following is an intuitive explanation of ”algorithm”, not a formal definition : *An algorithm is a set of instructions for some calculation that can be delegated to a machine*.
- Hence it must - be precise, unambiguous, - not allow subjective decisions.

Problem & Problem's instance

- **An algorithm solves some general problem, i.e. it works for a whole class of instances, not only for a particular one**.
- A problem is a set and a problem's instance is an element of this set. An algorithm solves all the problem's instances of a set.

Problem & Problem's instance

- Example, the Prime Problem : "Given a positive integer greater than one" (parameter), does this number can be divided by an integer other than one, "yes or no" (property to be satisfied by the answer) ?
- An **instance** of a problem is obtained by associating values to the formal parameters of the problem.
- An instance of Problem Prime is "Is 9 prime ?"

An every-day example of an algorithm

- Paper-and-pencil methods for multiplying two integers, as learned at school.
- Problem : Given two (arbitrary !) integers, compute their product.
- Instance : a pair of integers.

How to describe algorithms

By programs : Specific realization of a (more abstract) algorithm.

By pseudocode : Similar to usual procedural languages, free of non-essential details.

```
**SequentialSearch**$(\overbrace{A[1..n], val}^{input})$
**int** loc $= 1$;
**while** (loc $\leq$ n & A[loc] $\neq$ val)
        loc $=$ loc $+ 1$;
**return** $\overbrace{\text{loc}}^{output}$;
```

How to describe algorithms

Informally, in words. Programmer must fill in all details.

Given an array of integers

$$222, 179, 77, 156, 18, 96, 12$$

sort it in increasing order.

Algorithm :

- search for the smallest integer in the array and swap it with the integer in the first entry
- search for the second smallest integer in the array and swap it with the integer in the second entry

  $\vdots$
- search for the largest integer in the array and swap it with the integer in the last entry

How to describe algorithms

- search for the smallest integer in the array and swap it with the integer in the first entry
- search for the second smallest integer in the array and swap it with the integer in the second entry

  ⋮
- search for the largest integer in the array and swap it with the integer in the last entry

Same algo in pseudocode

```
**Selection sort**(A[1..n])
**for** $i = 1$ **to** $n - 1$ **do**
    $minj = i$ ; $minx = A[i]$ ;
    **for** $j = i + 1$ **to** $n$ **do**
        **if** $A[j] < minx$ **then**
            $minj = j$ ; $minx = A[j]$ ;
    A$[minj] = A[i]$ ; $A[i] = minx$ ;
```

Problems, Algorithms and Programs

- We design algorithms and programs for a problem, not for a problem instance. A given algorithm or program should work for all the problem's instances !
- There is *often* many different algorithms for a same problem (ex. multiplication : à la russe, Arabic, classic [American, English], divide-and-conquer)
- There is *always* many different programs for a same algorithm.

Algorithm analysis

- Given that there is often several algorithms for a same problem, one might want to compare the algorithms with each others
- If you discover a new algorithm, you might want to compare it with existing algorithms
- Then the question becomes :
  - By what should we compare two algorithms with each others ?
  - What should we use as a measure of how "good" an algorithm is ?
- Algorithm analysis is about comparing algorithms as well as making sure that algorithms solve the problem they have been designed to solve.

Algorithm behavior : running time

- The time taken by an algorithm to solve a problem instance is a critical component of an algorithm behavior
- But the running time for every instance is too complicated and cumbersome to calculate.

$$222, 179, 77, 156, 18, 96, 12$$

$$112, 189, 976, 551, 94, 18, 55$$

- Rather we compute the *running time on instances of a given input size*.
  - Consider maximum or average of running times for all instances of size n. (Worst-case/ average-case analysis.)
  - **Size** n must be defined appropriately : e.g. number of bits, digits, items.

Algorithm behavior : running time

- We need to be little bit careful when measuring the running time of an algorithm because other factors can affect the computational time. For example :
  - Characteristics of the computer (e.g. processor speed, amount of memory, file-system type, number and type of elementary operations, size of the registers, etc.)
  - The way the algorithm is coded (the program ! ! !)
  - The particular instance of data the algorithm is operating on (e.g., amount of data, type of data).

“Rate of growth”

- Show how time scale when problem instance size grow ?

$$222, 179, 77, 156, 18, 96, 12$$

$$222, 179, 77, 156, 18, 96, 12, 112, 189, 976, 551, 94, 18, 55$$

- Example : compare $n; n^2; 2^n$.
- The rate of grow ignores constant factors in running time as they depend on other factors than the algorithm itself : - speed of machine, - minor implementation details, etc.
- Using the rate of growth, we can compare algorithms, independent of the implementation details

Time taken for different running times and input sizes

$f(n)$	$n = 2$	$n = 64$	$n = 1024$	$n = 65536$
1	1 μs	1 μs	1 μs	1 μs
$\log_2 n$	1 μs	6 μs	10 μs	16 μs
n	2 μs	64 μs	1.02 ms	65.5 ms
$n \log_2 n$	2 μs	384 μs	10.2 ms	1.05 s
n^2	4 μs	4.1 ms	1.04 s	71.6 min
n^3	8 μs	0.26 s	17.9 min	8.9 yr
2^n	4 μs	5849 cen	$2^{972.5}$ cen	2^{65484} cen

Note : μs = microsecond = one millionth of a second, ms = millisecond = one thousandth of a second, s = second, min = minute, yr = year, and cen = century. (Assume 1 μs per op.)

Time is not “wall-clock” time

- *Elementary operations* are used to measure time complexity of algorithms.
  It is assumed that one elementary operation requires one time ”unit”.
- What is ”elementary”? Depends on type of data, like ”input size”.
- Some simple measure of the “size” of the data that the algorithm is operating on are needed, e.g., the size of an array, the number of nodes in a graph, the dimensions of a matrix.
- Some examples below show appropriate viewpoints.

Example

```
**SequentialSearch**(A[1..n], val)
loc = 0 ; (1 instr but 10 machine opts)
while (loc < n & A[loc] ≠ val) (1 instr but 200 machine opts)
    loc = loc + 1 ; (1 instr but 20 machine opts)
return loc ; (1 instr but 5 machine opts)
```

- What is the size of the input ?
- What are the elementary operations ?
- How many elementary operations, as a function of the array size n, are required by SequentialSearch ?

Example

```
**Selection sort**(A[1..n])
**for** $i = 1$ **to** $n - 1$ **do**
    $minj = i$ ; $minx = A[i]$ ;
    **for** $j = i + 1$ **to** $n$ **do**
        **if** $A[j] < minx$ **then**
            $minj = j$ ; $minx = A[j]$ ;
    A[$minj$] $= A[i]$ ; $A[i] = minx$ ;
```

- What is the size of the input ?
- What are the elementary operations ?
- How many elementary operations, as a function of the array size n, are required by Selection sort ?

"Analyzing the running time behavior"

- Identify typical input data
- Identify elementary operations
- Derive a mathematical analysis (exact analysis followed by the grow rate stated using asymptotic notation)
- Associate the algorithm to a *complexity class* (polynomial, exponential).

Typical input data

- We first need to determine what the input is, and *how much data* is being input.
- We need to determine which of the data affects the running time.
- We usually use n to denote the number of data items to be processed.
- This could be
  - size of a file
  - size of an array or matrix
  - number of nodes in a tree or graph
  - degree of a polynomial

Elementary operation

We talk about elementary operations (instructions) when we consider operations in a hardware independent fashion.

Recall that we are interested in rate of growth, not the exact running time. Thus, we can pick operations (instructions) that will run most often in the code.

Determine the number of times these operations will be executed as a function of the size of the input data.

Need to pick the operations that are executed most often, need to recognize when an operation can or cannot be performed in a constant amount of time.

Example : Sorting

- Problem statement : Sort in increasing order an array of n integers, output the sorted array.
- Algorithm :

```
**Selection Sort**(A[1..n])
for $i = 1$ to $n - 1$ do
    $minj = i$ ; $minx = A[i]$ ;
    for $j = i + 1$ to $n$ do
        if $A[j] < minx$ then
            $minj = j$ ; $minx = A[j]$ ;
    A[$minj$] $= A[i]$ ; $A[i] = minx$ ;
```

- We focus on the comparison ($<$) (this is the elementary operation) and ignore all the other instructions.

Example : Sorting

```
Selection Sort(A[1..n])
for $i = 1$ to $n - 1$ do
    $minj = i$ ; $minx = A[i]$ ;
    for $j = i + 1$ to $n$ do
        if $A[j] < minx$ then
            $minj = j$ ; $minx = A[j]$ ;
    A[$minj$] = $A[i]$ ; $A[i] = minx$ ;
```

- If we calculate the number of operations in this function based on the comparison operator, we have :
  - for $n = 1$, 0 comparison
  - for $n = 2$, 1 comparison

    $\vdots$
  - for n, $n^2/2$ comparisons

Example : Search for Maximum

- Problem statement : Search the maximum element in an array of n integers.
- Algorithm :

```
max(A[1..n])
max_value = int.MIN_VAL ;
for ( $i = 0; i < n; i++$ )
    max_value = MAXIMUM(max_value, A[i]) ;
return max_value ;
```

- We focus on the assignment (=) inside the loop and ignore the other instructions.
  - for an array of length 1, 1 comparison
  - for an array of length 2, 2 comparisons

    $\vdots$
  - for an array of length n, n comparisons