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

Lecturer: Michel Toulouse

Hanoi University of Science & Technology
michel.toulouse@soict.hust.edu.vn

9 mars 2020

Outline

- Definitions of terms like *algorithm*, *input size*, *formal parameters*, *output*, *running time*, *time 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.
- Algorithms on YouTube, see
  `https://www.youtube.com/watch?v=6hfOvs8pY1k`,
  `https://www.youtube.com/watch?v=e_WfC8HwVB8`,
  `https://www.youtube.com/watch?v=CvSOaYi89B4`

What is an algorithm ?

- An algorithm is a solution to a problem in a form of a sequence of instructions precise and non-ambiguous enough that they can be expanded into a program for running the solution onto a computer.
- Algorithms are usually described using pseudocode, which is similar to procedural languages, but free of programming languages syntactic details.
- The following is an example of pseudocode describing a sorting 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[$minj$] $= A[i]$; $A[i] = minx$;

Problem, Problem's instance & algorithms

A problem is a set and a problem's instance is an element of this set.

Example : the *Prime Problem* : "Given a positive integer (formal parameter) greater than one", does this number can be divided by an integer other than one, "yes or no"?

Instances of a problem are obtained by associating values to the formal parameters of the problem.

An instance of Problem Prime is "Is 9 prime ?"

An algorithm solves a problem, i.e. it works for the whole set of instances, not just for a few of them.

Problems, Algorithms and Programs

A given algorithm or program should work for all the problem's instances ! We design algorithms and programs for a problem, not for a problem instance.

There are *often* many different algorithms for a same problem (ex. sorting an array of integers : insertion, selection, merge sort, quicksort, bubblesort)

There is always many different programs for a same algorithm.

Many algorithms for a same problem

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[$minj$] = $A[i]$; $A[i] = minx$;

InsertionSort(A[1..n])
for $(i = 2; i <= n; i++)$
    v=A[i] ; j=i-1 ;
    while$(j > 0$ & $A[j] > v)$
        A[j+1]=A[j] ; j–;
    A[j+1] = v ;

Many algorithms for a same problem

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 :

- How should we compare two algorithms with each others ?
- What should we use as a measure of how "good" an algorithm is ?

There are different answers to these questions, we will study some of them

At the beginning of this course we will develop techniques for comparing algorithms with each other

Comparing running (execution) times

The running time to solve instances is a relevant attribute for comparing algorithms with each other

Pretty much every one prefer algorithms that run fast, the faster the better

But we need to be little bit careful when measuring the running time of an algorithm because factors not related to the algorithm can affect the running time.

Experimental Running Time

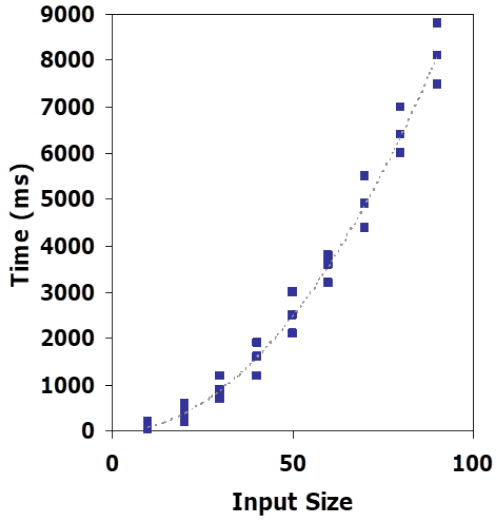
Write a program implementing the algorithm

Run the program with inputs of varying size and composition

Use a method like clock( ) to get an accurate measure of the actual running time

```
clock_t startTime = clock();
doSomeOperation();
clock_t endTime = clock();
clock_t clockTicksTaken = endTime - startTime;
double timeInSeconds = clockTicksTaken / (double) CLOCKS_PER_SEC;
```

Plot the results



Limitations of experimental running time measurements

Experimental evaluation of running time is very useful but

- It is necessary to code the algorithm, which may take some time
- Results may not be indicative of the running time on other inputs not included in the experiment
  - The particular instance of data the algorithm is operating on (e.g., amount of data [*input size*], type of data).
- In order to compare two algorithms, the same hardware and software environments must be used
  - 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 ! ! !)

Running time is not "wall-clock" time

To avoid the issues describe in the previous slide we don't measure running time using experimentally (wall-clock measurements)

Rather mathematical analysis is used to estimate the time needed to execute an algorithm.

Mathematical analysis of algorithms' running time

Uses a pseudo-code description of the algorithm instead of program code

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

- Changing the hardware/software environment affects the running time by a constant factor, but does not alter the growth rate of the running time

The first lectures of this class are dedicated to describe mathematical tools used to analyze the running time of algorithms

Identifying the input size

- We first need to determine what the input is, and *how much data* (the input size) is being input
- We usually use n to denote the input size
- We need to determine among the formal parameters which one impacts the running time. 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

```
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 ;
```

Basic operations

Basic or elementary operations are simple operations in the pseudo-code of an algorithm, such as assignments, elementary arithmetic operations, or loop control operations

It is assumed that one elementary operation requires one time "unit".

To estimate the time used by an algorithm, we count the number of time a particular elementary operation is executed in terms of the input size, the size of the problem instance.

Basic operations, a first example

The problem : search for a value *val* in an array $A[1..n]$ of n integers. If *val* is found returns the entry *loc* in A where *val* has been found, else returns $n + 1$.

Here is the pseudocode of an algorithm to solve this problem :

SequentialSearch(A[1..n], val)

loc $= 1$;	(*1 EO : 10 machine opts*)
while (loc $\leq$ n & A[loc] $\neq$ val)	(*1 EO : 40 machine opts*)
loc $=$ loc $+ 1$;	(*1 EO : 20 machine opts*)
return loc ;	(*1 EO : 5 machine opts*)

- What are the elementary operations ?
- Which EO will be a good one to measure the running time of this algorithm ?
- How many elementary operations are required by SequentialSearch ?

Basic operations, a second example

The problem : Given an array $A[1..n]$ of n integers, sort the elements of the array in increasing order.

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 are the elementary operations ?
- Which EO will be a good one to measure the running time of this algorithm ?
- How many elementary operations are required by Selection sort ?

second example (continue)

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[$minj$] = $A[i]$; $A[i] = minx$;

Counting the number of basic operations : When $i = 1$, $n - 1$ basic operations are executed, when $i = 2$, ...

outer loop index	1	2	3	$\cdots$	n-2	n-1
# of basic ops	n-1	n-2	n-3	$\cdots$	2	1

so, the number of basic operations is
$1 + 2 + \cdots + n - 3 + n - 2 + n - 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2-n}{2}$

Running time & worst case instances

For many algorithms, we still cannot tell their running time just by counting the number of times an elementary operation is executed

Example, SequentialSearch, for the following two problem instances of same size : $val = 4, A = [1, 2, 3, 4]$ and $val = 4, A = [4, 3, 2, 1]$, the running time is different :

- SequentialSearch([1,2,3,4]) is 4
- SequentialSearch([4,3,2,1]) is 1

though again the input size of each problem instance is the same

To solve this issue we express the running time for the worst case instance of a given size

Worst case analysis

Same as before except one needs to find the worst instances, i.e. the instances which for a given size n have the largest possible running time

This is called worst-case analysis of the running time.

There are also two other types of analysis :

- There is also best-case analysis where one seeks to find the instances with the lowest running time
- and average-case analysis, average-case will be described in later in another lecture

Worst case running time analysis

```
SequentialSearch(A[1..n], val)
loc = 1 ;
while (loc ≤ n & A[loc] ≠ val)
    loc = loc + 1 ;
return loc ;
```

- What is (are) the worst-case instance(s) of a given size n
- How many elementary operations, as a function of the array size n, are required in worst case ?
- What is the best-case instance ?

Selection sort, worst case analysis

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 worst case ?

Selection sort, worst case analysis

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 worst case ?

In fact there is no worst case (or all instances are worst case).

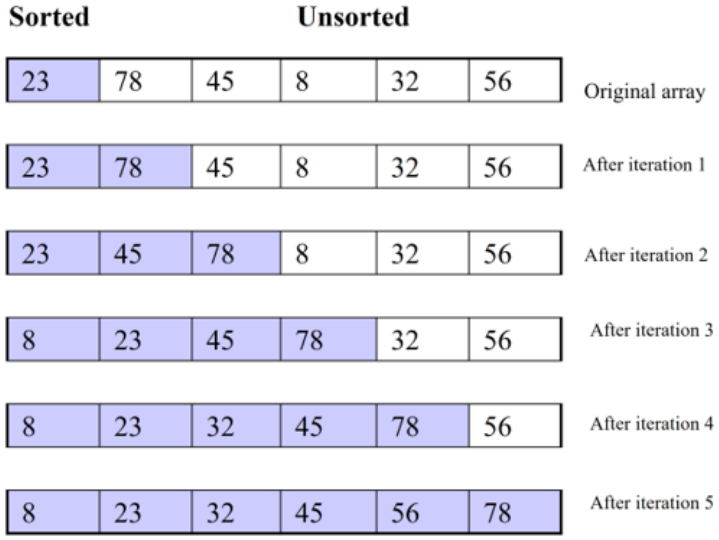
The execution of the two loops is independent of how the input array is structured, therefore the running time for selection sort is the same for all inputs of a given size n

Insertion sort, worst case analysis

InsertionSort(A[1..n])
for $(i = 2; i <= n; i++)$
    v=A[i] ; j=i-1 ;
    while$(j > 0$ & $A[j] > v)$
        A[j+1]=A[j] ; j–;
    A[j+1] = v ;

Running time

for $i = 2$ one basic operation is executed

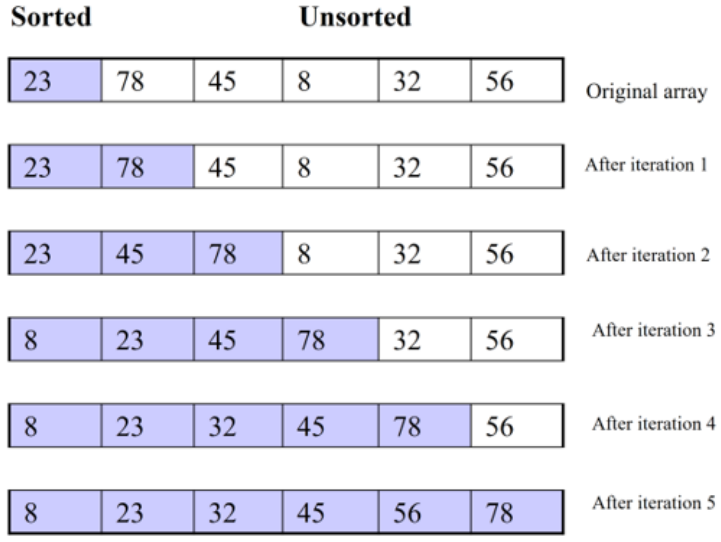


Insertion sort, worst case analysis

InsertionSort(A[1..n])
for $(i = 2; i <= n; i++)$
    v=A[i] ; j=i-1 ;
    while$(j > 0$ & $A[j] > v)$
        A[j+1]=A[j] ; j–;
    A[j+1] = v ;

Running time

for $i = 3$, two basic operations are executed,therefore $1 + 2$

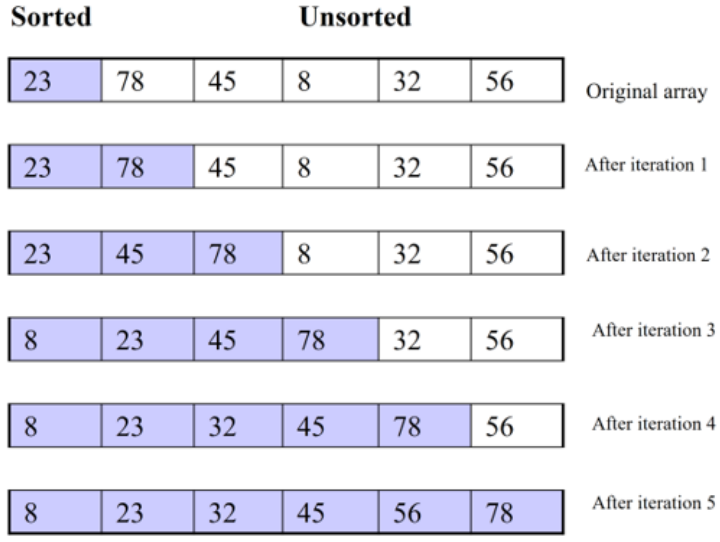


Insertion sort, worst case analysis

InsertionSort(A[1..n])
for $(i = 2; i <= n; i++)$
    v=A[i] ; j=i-1 ;
    while$(j > 0$ & $A[j] > v)$
        A[j+1]=A[j] ; j–;
    A[j+1] = v ;

Running time

for $i = 4$, three basic operations are executed,therefore 1 + 2 +3

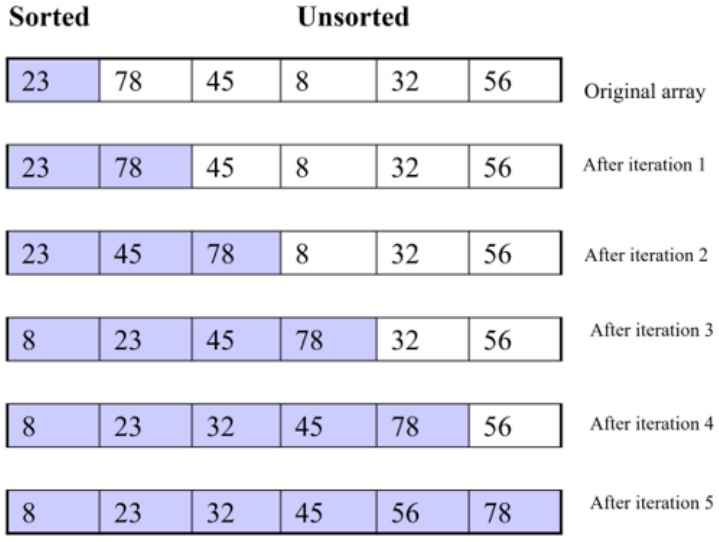


Insertion sort, worst case analysis

InsertionSort(A[1..n])
for $(i = 2; i <= n; i++)$
    v=A[i] ; j=i-1 ;
    while$(j > 0$ & $A[j] > v)$
        A[j+1]=A[j] ; j–;
    A[j+1] = v ;

Running time

for $i = 5$, two basic operations are executed,therefore $1 + 2 + 3 + 2$



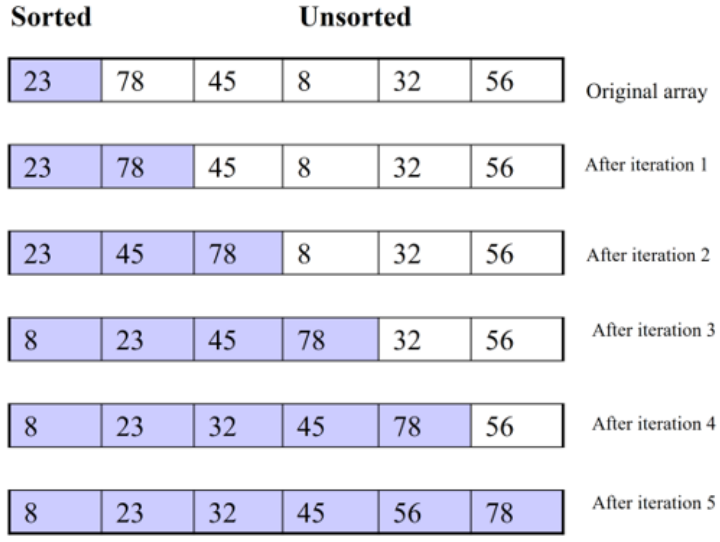
Insertion sort, worst case analysis

InsertionSort(A[1..n])
for $(i = 2; i <= n; i++)$
    v=A[i] ; j=i-1 ;
    while$(j > 0$ & $A[j] > v)$
        A[j+1]=A[j] ; j–;
    A[j+1] = v ;

Running time

for $i = 6$, one basic operation is executed,therefore

$1 + 2 + 3 + 2 + 1 = 9$ basic operations



Insertion sort, worst case analysis

The input array is already sorted (best case)

InsertionSort(A[1..n])
for $(i = 2; i <= n; i++)$
    v=A[i] ; j=i-1 ;
    while$(j > 0$ & $A[j] > v)$
        A[j+1]=A[j] ; j–;
    A[j+1] = v ;

8	23	32	45	56	78
8	23	32	45	56	78
8	23	32	45	56	78
8	23	32	45	56	78
8	23	32	45	56	78
8	23	32	45	56	78

Running time

In this case the running time is $n - 1$

Insertion sort, worst case analysis

The input array is sorted in decreasing order (worst case)

InsertionSort(A[1..n])
for $(i = 2; i <= n; i++)$
    v=A[i] ; j=i-1 ;
    while$(j > 0$ & $A[j] > v)$
        A[j+1]=A[j] ; j–;
    A[j+1] = v ;

Running time

In this case the running time is $2 + 3 + 4 + 5 + 6 = \sum_{j=1}^{n} -1 = \frac{n(n+1)}{2} - 1$

78	56	45	32	23	8
56	78	45	32	23	8
45	56	78	32	23	8
32	45	56	78	23	8
23	32	45	56	78	8
8	23	32	45	56	78

Example : Search for Maximum

Problem : 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 ;
```

Selecting (=) inside the loop as EO, the worst-case running time in terms of n are :

- for an array of length 1, 1 comparison
- for an array of length 2, 2 comparisons

  ⋮
- for an array of length n, n comparisons

Here worst-case and best-case are the same.

Mathematical analysis, what's next

Counting the number of time a basic operation executed is not always trivial. We will study some patterns that often occur in algorithms, recall mathematical summations and their closed forms which apply to these patterns

While comparing algorithms with each other we are not interested in the exact number of basic operations executed but rather we focus on the number of operations in terms of the input size.

We are interested in the growth rate of the running time of algorithms, we will describe the techniques used in algorithmic to classify algorithms according to their grow rate.