Design and Analysis of Algorithms

Jun Han

Prof. of Computing Science

BUAA

Course Overview

Administrative Details

Staff

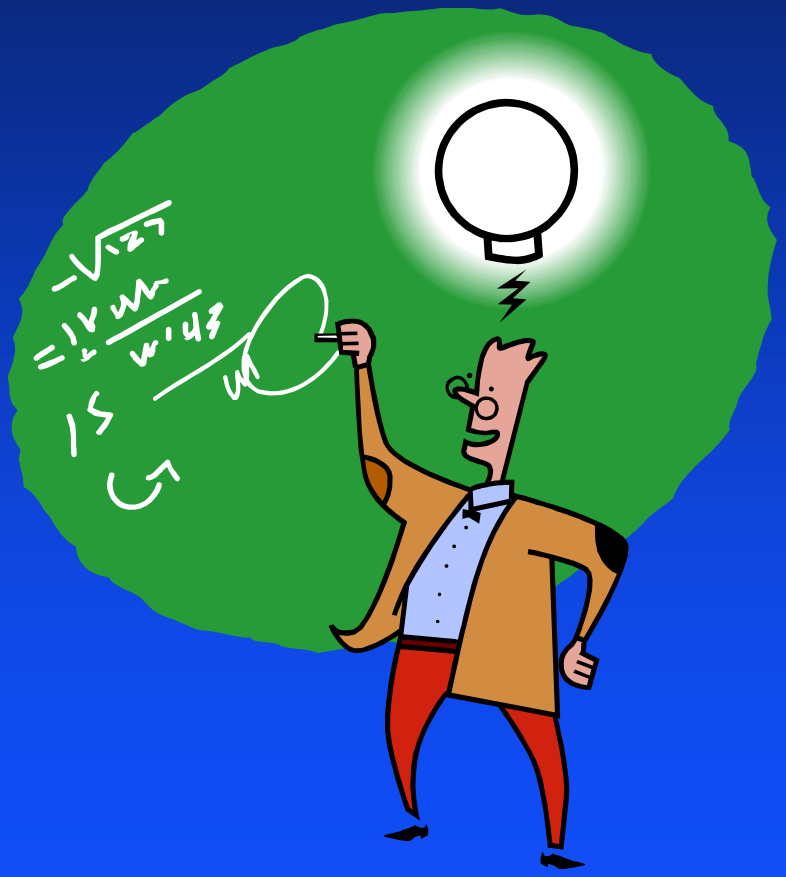
Lecturer:
韩军

Location:
北航新主楼 G 座 1112.

Phone: 8231 6340

Email:
jun_han@buaa.edu.cn

Profile:
http://www.act.buaa.edu.cn/

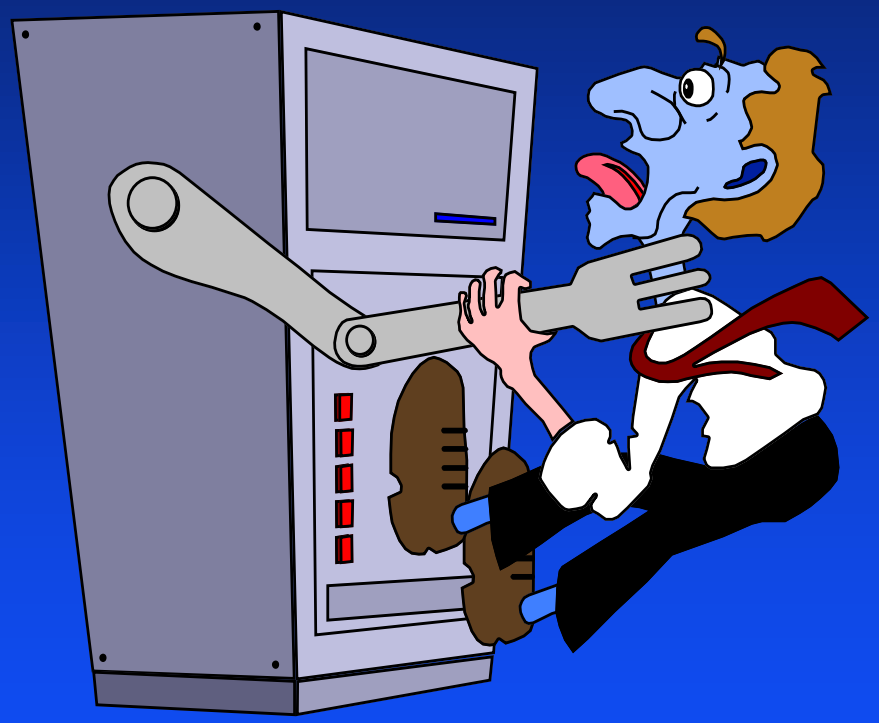


Your background

I assume that you have substantial programming skills and

Knowledge of

Data structure



You need to be comfortable with basic concepts from Discrete Maths.

About Algo. ? Assumed avg

You Are Supposed to Be

Cool minded

Self motivated

Programming addicted ?

AND

Your feedback is Valuable

Course Representative and Tutors

To Be Advised

Course Outline

This is NOT a first course in Algo. design and analysis

It is intended to provide the basis for competent design and programming algorithms with optimality

You need to acquire not only theory but skills in designing, analysing and programming algorithms

Syllabus

Chapter 1	Introduction
Chapter 2	Complete Development of an Algorithm
Chapter 3	Sorting，Searching and Matching Algo.s ？？
Chapter 4	Divide and Conquer
Chapter 5	Dynamic Programming
Chapter 6	Greedy Algorithms
Chapter 7	Backtracking
Chapter 8	Branch and Bound
Chapter 9	NP Completeness (& Lower Bound Argument)
Chapter 10	Approximation Algorithms
Chapter 11	Randomized Algorithm
Chapter 12	Heuristics: GA, NS, SA, TS, ACO, PSO

Sorting

Bubble sort
Linear insertion sort
Quicksort
Shellsort
Heapsort
Linear probing sort

Merge sort
Bucket sort
Radix sort
Hybrid methods
Treesort

Balanced merge sort
Cascade merge sort
Polyphase merge sort
Oscillating merge sort
External quicksort

List merging
Array merging
Minimal-comparison merging

Searching

Sequential search
Basic Sequential search
Self-organizing Sequential search
Optimal Sequential search
Jump search

Sorted array search
Binary search
Interpolation search
Interpolation-Sequential search

Binary tree search
B-trees
Index and indexed sequential files
Digital trees

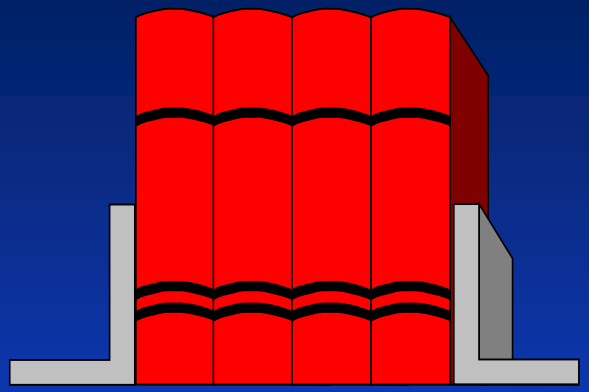
Multidimensional search
Quad trees
K-dimensional trees

Hashing
Uniform probing hashing
Random probing hashing
Linear probing hashing
Double hashing
Quadratic hashing
Ordered hashing
Reorganization for Uniform probing: Brent's algorithm
Reorganization for Uniform probing: Binary tree hashing
Optimal hashing
Direct chaining hashing
Separate chaining hashing
Coalesced hashing
Extendible hashing
Linear hashing
External hashing using minimal internal storage

Syllabus

Chapter 1	Introduction
Chapter 2	Complete Development of an Algorithm
Chapter 3	Sorting，Searching and Matching Algo.s ？？
Chapter 4	Divide and Conquer
Chapter 5	Dynamic Programming
Chapter 6	Greedy Algorithms
Chapter 7	Backtracking
Chapter 8	Branch and Bound
Chapter 9	NP Completeness (& Lower Bound Argument)
Chapter 10	Approximation Algorithms
Chapter 11	Randomized Algorithm
Chapter 12	Heuristics: GA, NS, SA, TS, ACO, PSO

Reference Books



Algorithms
R. Sedgewick

Analysis of Algorithms
R. Sedgewick and P. Flajolet；

Introduction to Algorithms
Thomas Cormen and Charles E. Leiserson

The Design and Analysis of Computer Algorithms
Alfred Aho，Hopcroft，Ullman

《算法设计与分析》
周培德

Languages (Lectures and Notes)

Mainly Chinese and some English

Why English?

- Consistency;
- Knowledge;
- English in computer science;
- Your future work.

* I will suppose you can understand;

Languages (Programming)

Whatever you like

C++
Java
Pascal
Delphi
VB
……

Woo!

Home Work and Assessment Overview

During the semester there will be 3 home work question sheets

and 3 programming assignments to help you to monitor your progress.

Plan to work **consistently!**

Your regular work gives me valuable evidence that you can design and code an algorithm.

Assessment

Marks from these components and from your final examination will be combined to produce a single mark as follows:

- –30% (3 x 10%) for home works
- –30% (3 x 10%) for assignments
- –40% for the final examination
  (written, close book, close note)

Attendance – 0 to -10

General instructions for assignments

Please note very carefully the following points (applicable to ALL assignments). If you are not sure of the exact requirements, please ask.

- Include a comment at the start of your program indicating your name and student number.
- Include complete pseudo code and a flow chart of your program.
- Choose identifier names very carefully to convey the use of those identifiers.
- Ensure that the layout of your program statements follows a sensible indentation convention.
- Submit **Source code** and **executable program**

Marks may be lost if you do not follow these requirements.

Assignment deadlines

You are expected to schedule your work to meet deadlines - this is what happens in the "real world".

Also, it is simply not fair to other students who in most cases will have worked very hard, and perhaps have given up leisure activities in order to meet deadlines.

Assignments: general advice

For all assignments, use at least the given test data.

Your program will be tested with this and some other data and it must behave at least as well as my version, which is by no means perfect.

Examine the sample output for each program.

From this it should be clear that you need to check for various error conditions.

Getting to 1st base

If your program fails to compile you will receive zero credit.

Please also ensure that it runs as per the specification given.

If you are not sure as to the exact requirements, please ask.

More Legalities

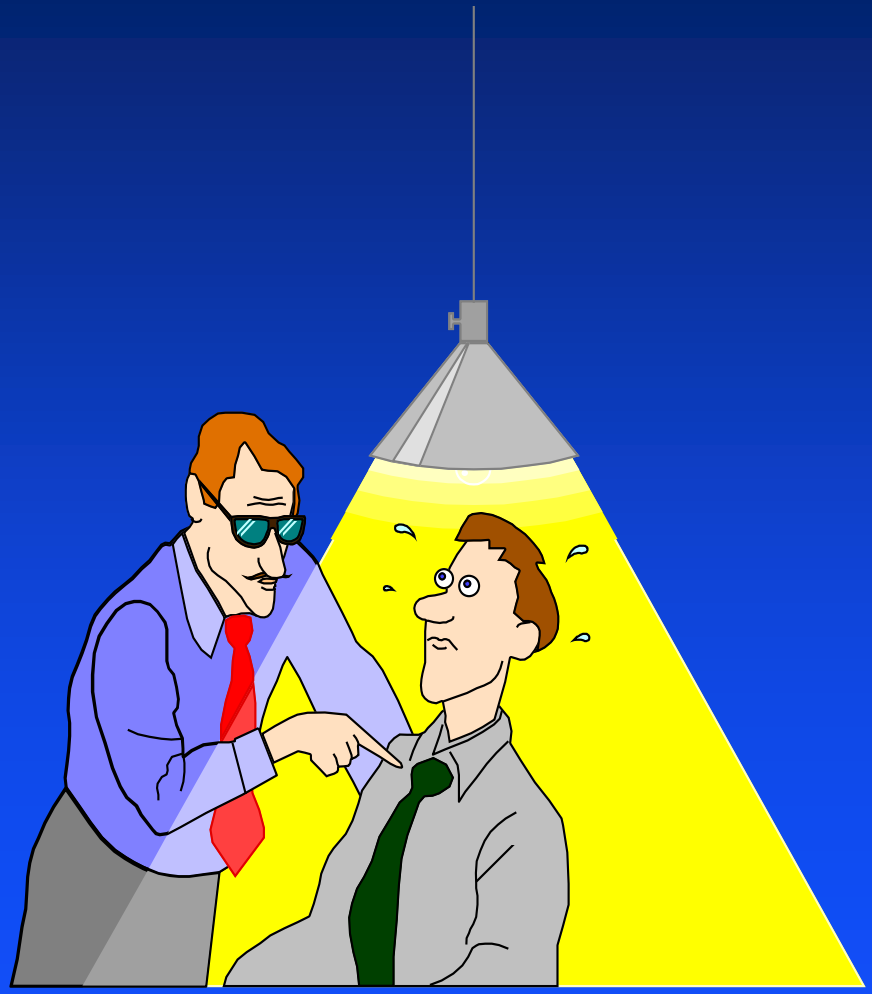
For this subject, as in all Computing Science subjects except where group assignments are organized, assignments must be substantially your own work.

Submissions will be routinely scrutinized for similarity.

Originality

It is perfectly in order to discuss what the requirements are, or even to discuss your approach, but code must be your own for at least two reasons.

- –You need the (sometimes arduous) experience of writing programs.
- –We need to test your skills.

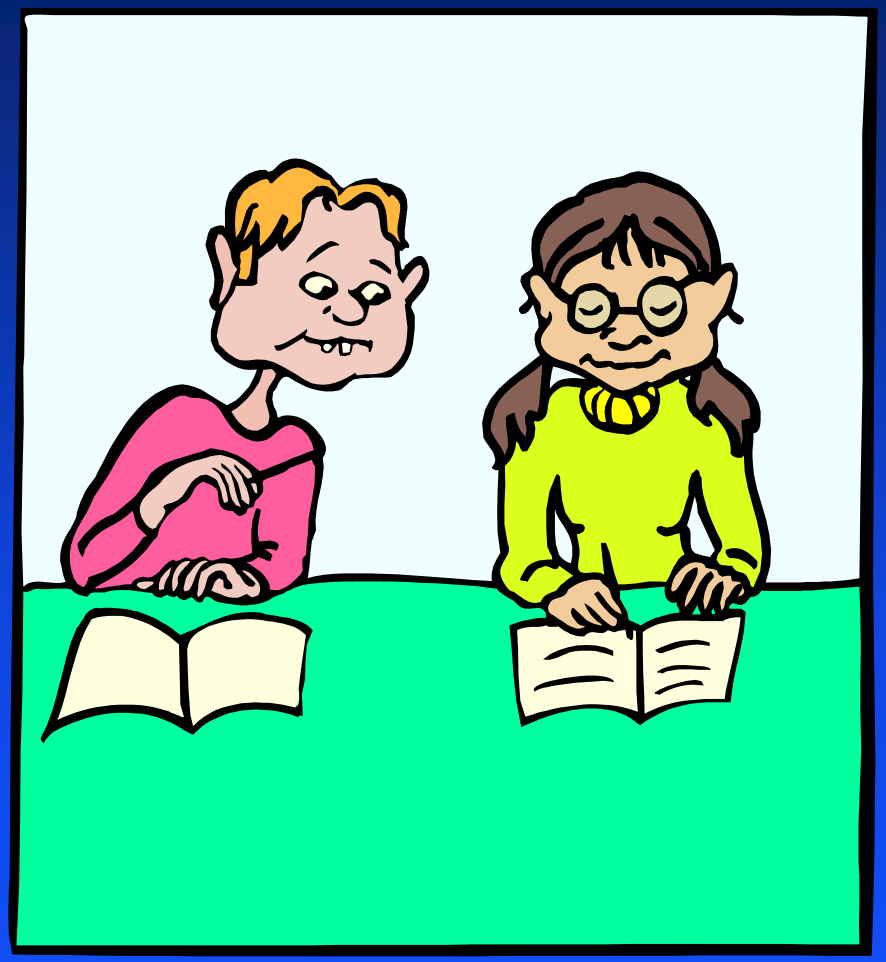


Plagiarism

If you copy from someone, or work very closely with a friend, we will probably detect it.

When we do, you will be penalized.

You should familiarize yourself with the university rules regarding plagiarism.



More Tips

PPT slides

--- source

--- preview

--- in class use & laser pointer

--- after class

Classroom interact

Unclear parts

Teaching Group

Anything left ?

Design and Analysis of Algorithms

Introduction

What is an algorithm ?

Algorithms

An algorithm is a set of instructions for performing a task.

It must have the following features:

- Each instruction must be clear and executable.
- The next step must be determinate.
- The process must terminate.
- The goal must be achieved.

Examples of algorithms are:

- a recipe.
- a musical score.
- a knitting pattern.

Features of Algorithms

Each algorithm has an author and a static form.

Upon execution by a processor, we have a dynamic process.

The processor for a recipe is a cook, for a musical score is a performer.

Algorithms: Declarations

Objects are manipulated.

Programs operate on data.

Instructions may be preceded by a declaration of the objects involved.

This is true of recipes, model aeroplane assembly

It is also true of many modern programming languages.

Algorithms: Decisions

Instructions are executed sequentially.

Decisions may be required depending on the current situation (conditional instruction).

Algorithms: Repetition

Some parts of the algorithm may be executed many times **(iteration)**.

The number of repetitions may be specified directly (knit five rows) or by testing for a desired state (cook until the surface is light brown).

Algorithms: Procedures

A group of instructions may be replaced by a single instruction.

The group is called a procedure or method

It is actually a sub-algorithm (solves part of our problem).

We shall see the great importance of being able to solve small parts of problems as sub-problems.

Program = Algorithm + Data

A program is a collection of algorithms expressed in a language which a computer can use.

Before constructing a program, we must be able to answer the following two questions:

- What processing is required?
- What data are to be processed?

The program often will **input** some required values, **process** these in the correct manner and then **output** the results to the user.

Algorithm Classifications and Course Coverage

- (Non-) Numerical
- Strategically
- Complexity
- Exactness
- (Non-) Deterministic
- Application Orientated
- ……

Why analyzing an algorithm?

Why

We analyze algorithms with the intension of

Improving them

And for

Choosing

among several available Algo.s for a problem

How to analyze an algorithm?

Analyzing an algorithm

We will use the following criteria:

- Correctness
- Amount of work done
- Amount of space used
- Simplicity
- Optimality

Correctness

-----An algorithm is correct if when given a valid input it computes for a finite amount of time and produces the right answer.

对于每一个合法输入，算法都会在有限的时间内输出一个满足要求的结果

This definition is helpful if we know what the valid inputs are and what “the right answer” is.

Thus showing that an algorithm is correct means making a precise statement about what result it is to produce when given certain specified inputs, and then proving the statement.

Correctness

There are two aspects to an algorithm: the problem solution method and the sequence of instructions for carrying it out.

Establishing the correctness of the method and/or formulas used may require a long sequence of lemmas and theorems about the objects on which the algorithm works (for example, graphs, permutations, matrices).

Correctness

If an algorithm is fairly short and straightforward, we generally use some very informal (and indescribable) means of convincing ourselves that the various parts do what we expect them to do.

We may check some details carefully (e.g., initial and final values of loop counters), and hand-simulate the algorithm on a few small examples.

None of this proves that it is correct, but informal techniques may suffice for small programs.

Correctness

Most programs written outside of classes are very large and very complex.

To prove the correctness of a large program, we can try to break the program down into smaller segments, so that if all of the smaller segments do their jobs properly, then the whole program is correct, and then prove that each of the segments is correct.

This task is made easier if (it may be more accurate to say: “This task is possible only if”) algorithms and programs are written so that they can be broken down into disjoint segments that can be verified separately.

Correctness

One of the most useful techniques for rigorous proofs of correctness is mathematical induction.

It is used to show that loops in an algorithm do what they are intended to do.

For each loop we state conditions and relationships that we believe are satisfied by the variables and data structures used, and then verify that these conditions hold by inducting on the number of passes through the loop.

Example

A Sequential Search algorithm:

To find the location, or index, of a given item X in a list.

The algorithm compares X to each list entry in turn until a match is found or the list is exhausted.

If X is not in the list, the algorithm returns 0 as its answer.

Sequential Search

Algorithm SEQUENTIAL SEARCH
Input: *L, n, X* where *L* is an array with *n* entries.
Output: *j*.

1. $j \leftarrow 1$
2. **while** $j \leq n$ and $L(j) \neq X$
3. **do** $j \leftarrow j+1$ **end while do**
4. **if** $j > n$ **then** $j \leftarrow 0$

Correctness of Sequential Search

Before trying to show that the algorithm is correct, we should make a precise statement about what it is intended to do.

Correctness of Sequential Search

We may state that if *L* is a list with *n* entries, the algorithm terminates with *j* set to the index of the list entry equal to *X*, if there is one, and set to 0 otherwise.

如果 *L* 为一具有 *n* 个元素的列表, 算法终止 时 *j* 被赋值为列表中等于*X* 的元素（如果存在等于*X* 的元素）的索引值, 或被赋值为0 （如果不存在等于*X* 的元素）。

1. j ← 1
2. while j ≤ n and L(j) ≠ X
3. do j ← j+1 end while do
4. if j ＞ n then j ← 0

Correctness of Sequential Search

It does not specify the result if *X* appears more than once in the list, and

it does not indicate for which values of *n* the algorithm is expected to work.

We may assume that *n* is nonnegative, but what if the list is empty, i.e., if *n* = 0? A more precise statement is:

Correctness of Sequential Search

Given an array L containing n items ($n \geq 0$) and given X, the sequential search algorithm terminates with j set to the index of the first occurrence of X in L, if X is there, and set to 0 otherwise.

给定 X 及具有 n 个元素的数组L ($n \geq 0$), 算法终止 时 j 被赋值为列表中第一个等于X 的元素（如果存在等于X 的元素）的索引值, 或被赋值为0 （如果不存在等于X 的元素）。

Correctness of Sequential Search

We will prove this statement by proving a stronger one in a form to which we can apply induction:

For $1 \le k \le n+1$, if and when control reaches the tests in line 2 for the k^{th} time, the following conditions are satisfied:

1. $j = k$ and for $1 \le i < k$, $L(i) \ne X$.
2. If $k \le n$ and $L(k) = X$, the algorithm will terminate with j still equal to k after executing the tests and line 4.
3. If $k = n + 1$, the algorithm will terminate with $j = 0$ after executing the tests and line 4.

Correctness of Sequential Search

Proof

Let k = 1.

Then j = k from line 1; the second part of condition 1 is vacuously satisfied.

For condition 2, if 1 ≤ n and L(1) = X, the test in line 2 fails and control passes to line 4 where, since j = k ≤ n, j is unchanged.

For condition 3, if k = n + 1, then j = n + 1 so the test in line 2 fails and control passes to line 4 where j is set to zero.

(Note that this is the case where n = 0 and the list is empty.)

Sequential Search

1. $j \leftarrow 1$
2. while $j \leq n$ and $L(j) \neq X$
3. do $j \leftarrow j+1$ end while do
4. if $j > n$ then $j \leftarrow 0$

Correctness of Sequential Search

Now we assume that the three conditions are satisfied for some $k < n + 1$ and show that they hold for $k + 1$.

Suppose control has reached the tests in line 2 for the $(k+1)^{th}$ time.

Condition 1 for k and the fact that line 3 was executed once more before returning to line 2 for the $(k + 1)^{th}$ time imply that now $j = k + 1$.

Condition 1 for k says that $L(i) \neq X$ for $1 \leq i < k$.

Condition 2 for k implies that $L(k) \neq X$ since otherwise the algorithm would have terminated already; thus for $1 \leq i < k + 1$, $L(i) \neq X$ and condition 1 is established for $k + 1$.

Correctness of Sequential Search

1. $j = k$ and for $1 \leq i < k$, $L(i) \neq X$.

2. If $k \leq n$ and $L(k) = X$, the algorithm will terminate with j still equal to k after executing the tests and line 4.

3. If $k = n + 1$, the algorithm will terminate with $j = 0$ after executing the tests and line 4.

Correctness of Sequential Search

For condition 2, if j=k+1≤ n and L(j=K+1) = X, the test in line 2 fails and control passes to line 4 where, since j = k +1≤ n, j is unchanged.

For condition 3, if k+1 = n + 1,

then j =k+1= n + 1 so the test in line 2 fails and control passes to line 4 where j is set to zero.

```
1.      j ← 1
2.      while j ≤ n and L(j) ≠ X
3.              do j ← j+1 end while do
4.      if  j > n then j ← 0
```

Correctness of Sequential Search

The claims we have proved show :

- –that the tests in line 2 are executed at most $n + 1$ times,
- –that the output is $j = 0$ if and only if they were executed $n + 1$ times, and in that case for $1 \leq i < n + 1$, $L(i) \neq X$; i.e., X is not in L.
- –The output is $j = k$ if and only if $L(k) = X$ and for $1 \leq i < k$, $L(i) \neq X$ so k is the index of the first occurrence of X in the array.

Thus the algorithm is correct!

Correctness

If this proof seems a bit tedious, imagine what a proof of correctness of a full-sized program with complex data and control structures would be like.

But if one wants to rigorously verify that a program is correct, this is the sort of work that must be done!

Analyzing an algorithm

We will use the following criteria:

- Correctness
- Amount of work done
- Amount of space used
- Simplicity
- Optimality

Amount of work down

Efficiency?

Machine
and
Input

Number of basic operations done

Amount of work down

Size of Input

We can use a function $T(n)$ to represent the number of time units (basic operation here) taken by a program on any input of size n.

For the sake of convenience. we may assume that one time unit is equivalent to one programming statement.

Amount of work down

The running time (number of basic operations) of a program often also depends on the nature of the input as well as the size of the input.

As such, we can further classify running time in terms of worst, average and best case.

Tworst(*n*) is easier to compute than *Tavg*(*n*) though this measure may have more practical significance.

Tbest(*n*) is rarely used.

Amount of work down

To see how we can calculate $T(n)$, consider the following program fragment that determines the largest element of an array:

```
max=a[0];
For (i=1; i<n; i++)
{
If (a[i]>max)
max=a[i];
}
```

Amount of work down

we will assume that assignment statements (i.e. max=a[0];)

and testing statements (i.e. the if) take one time unit,

while the for loop uses three time units (one for each component of initialization, test and increment)

```
max=a[0];
for (i=1; i<n; i++)
{
   if (a[i]>max)
   max=a[i];
  }
```

Amount of work down

To calculate the $T(n)$, we do the following:

- 1 time unit for the initialisation of max.
- 1 time unit for the initialisation of the for loop.
- 1 time unit for the initial for loop test (i.e. i<n;)
- 2 time unit for the body of the loop

```
max=a[0];
for (i=1; i<n; i++)
{
   if (a[i]>max)
   max=a[i];
   }
```

Amount of work down

Worst case: In the worst case, for each iteration of the inner body of the loop we will perform both statements.

This would occur if a={1,2,3,4,5,6,7} for example. Therefore, for every iteration we would have 4 time steps (1 for the loop test, 1 for the increment, 1 for the if and 1 for the assignment).

This equates to 4(*n*-1).

```
max=a[0];
for (i=1; i<n; i++)
{
   if (a[i]>max)
   max=a[i];
   }
```

Amount of work down

Average case: For the average case, it would be reasonable to assume that every second element or so would be made the new max while the loop was iterating.

This would occur if a={5,2,7,4,8,6,9} for example.

In this case, we will still need 3 units of each iteration of the loop (2 for loop control and 1 for the if) and(n-1)/2 for the max assignment.

This equates to 3(*n*-1) +(n-1)/2 =3.5(*n*-1).

```
max=a[0];
for (i=1; i<n; i++)
{
   if (a[i]>max)
   max=a[i];
   }
```

Amount of work down

Best case: In the best case, a[0] would contain the maximum element.

Therefore, the program would never execute the max assignment statement.

This equates to 3(*n*-1).

```
max=a[0];
for (i=1; i<n; i++)
{
   if (a[i]>max)
   max=a[i];
  }
```

Amount of work down

From the above discussion we can see that:

- $Tworst(n) = 4(n-1) + 3 = 4n - 4 + 3 = 4n - 1$
- $Tavg(n) = 3.5(n-1) + 3 = 3.5n - 3.5 + 3 = 3.5n - 0.5$
- $Tbest(n) = 3(n-1) + 3 = 3n - 3 + 3 = 3n$

Time Complexity

Mathematically :

Average case

Let D_n be the set of inputs of size n for the problem under consideration.

Let I be an element of D_n and let p(I) be the probability that input I occurs.

Let t(I) be the number of basic operations performed by the algorithm of input I. Then its average behavior may be realistically defined as

$$A(n) = \sum_{I \in D_n} p(I) \bullet t(I)$$

Time Complexity

t(I) is to be computed by careful examination and analysis of the algorithm, but p(I) cannot be computed analytically.

The function p must be determined from experience and/or special information about the application for which the algorithm is to be used;

it is usually not easy to determine.

Time Complexity

Worst case:

$$w(n) = \max_{I \in D_n} t(I)$$

W(n) is the maximum number of basic operations performed by the algorithm on any input of size n. It can usually be computed more readily than A(n) can.

Example

A Sequential Search algorithm:

To find the location, or index, of a given item X in a list.

The algorithm compares X to each list entry in turn until a match is found or the list is exhausted.

If X is not in the list, the algorithm returns 0 as its answer.

Example

Algorithm SEQUENTIAL SEARCH

Input: *L*, *n*, *X* where *L* is an array with *n* entries.

Output: *j*.

1. $j \leftarrow 1$
2. **while** $j \leq n$ and $L(j) \neq X$
3. **do** $j \leftarrow j+1$ **end while do**
4. **if** $j > n$ **then** $j \leftarrow 0$

Example

Average-behavior analysis:

The inputs for this problem can be categorized according to where in the list X appears, if it appears at all.

That is, there are $n + 1$ inputs to consider.

For $1 \leqslant i \leqslant n$, let I_i represent the case where X is in the i^{th} position in the list and let I_{n+1} represent the case where X is not in the list at all.

Then, let $t(I)$ be the number of comparisons done (the number of times the condition $L(j) \neq X$ in line 2 is tested) by the algorithm on input I.

Obviously, for $1 \leq i \leq n$, $t(I_i) = i$, and $t(I_{n+1}) = n$.

Example

To compute the number of comparisons done on the average, we must know how likely it is that X is in the list and how likely it is that X is in any particular position.

Let q be the probability that X is in the list, and assume that each position is equally likely.

Then for $1 \leq i \leq n$,

$p(I_i)=q/n$

and $p(I_{n+1})=1-q$.

Example

Average-case analysis:

Example

Worst-case analysis:

Clearly $W(n) = \max\{t(I_i): 1 \le i \le n + 1 \} = n$.

The worst cases occur when X is the last entry in the list and when X is not in the list at all.

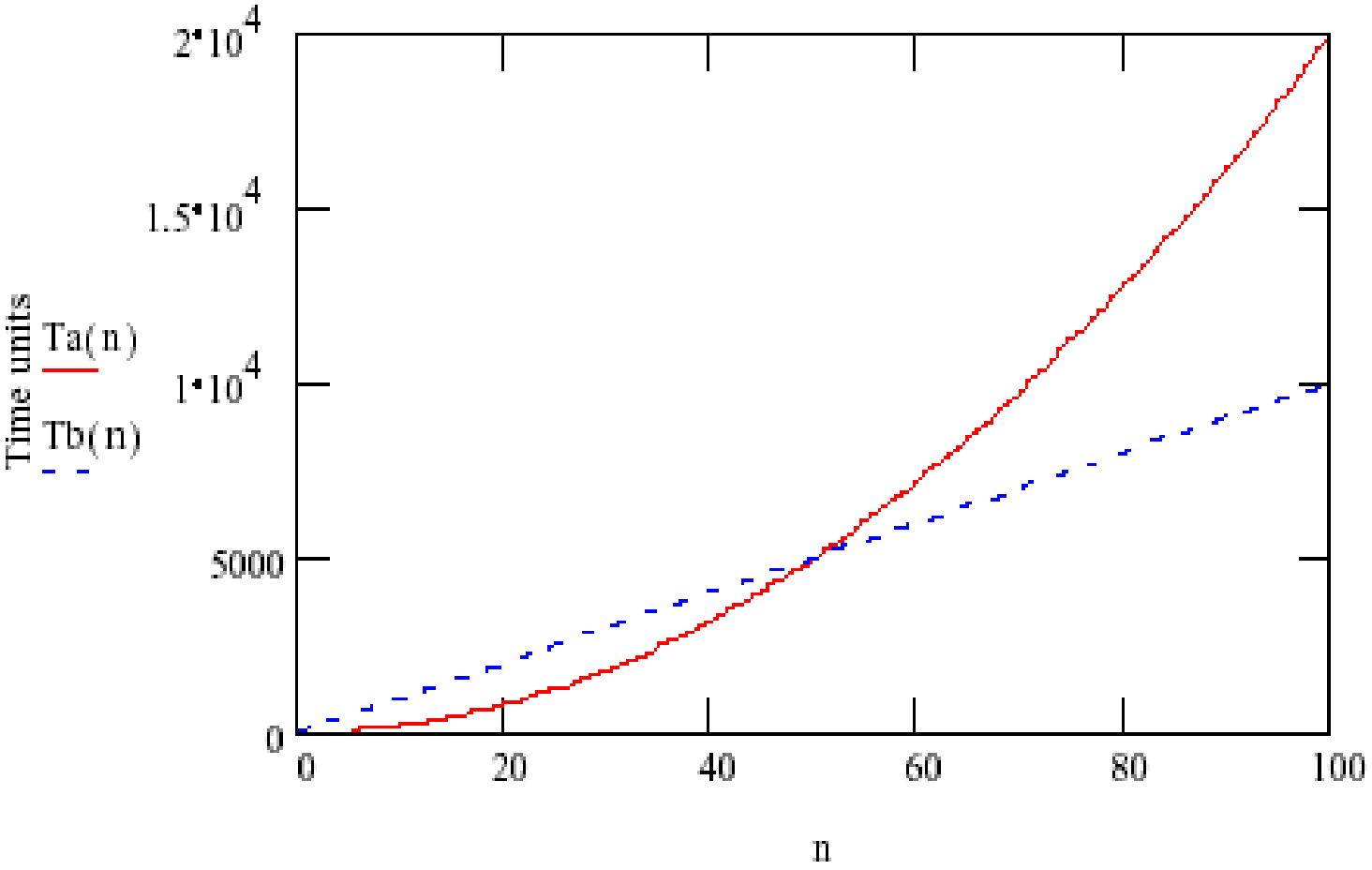
Comparing Algorithms on $T(n)$

Now that we have basis on which to calculate the number of time steps, we can compare two or more competing algorithms to decide which one we are likely to use and under what circumstances.

For instance, consider $T_a(n) = 2n^2$ and

$T_b(n) = 100n$.

$2n^2$ and $100n$



Approximate Running Time

The use of $T(n)$ neglects the fact that each statement will take a different amount of time depending on the compiler/interpreter and the hardware platform on which it is run.

Therefore, we cannot take a program and say that it will take *x* seconds to run, unless we know which machine and compiler/interpreter will be used (as well as the internal workings of the two).

To do this accurately is a near impossible task.

O (Big-Oh)

Big-oh allows us to ignore constants when evaluating the running time of a program.

For instance, in our previous example , we could say the program has $O(n)$ time complexity instead of $T(n) = 4(n-1)$, as the 4 and -1 are mere approximations.

The big-oh notation allows us to say “some constant multiplied by *n*, plus or minus another constant”.

O (Big-Oh)

To further refine our notion of big-oh, let us first make the following assumptions that $n >= 0$ and $T(n) >= 0$. Given that $f(n)$ is some function defined over n, we can say the "$T(n)$ is $O(f(n))$ if $T(n)$ is at most a constant times larger than $f(n)$, except possibly for some small values of n.

To refine this, we say that $T(n) <= cf(n)$ if there exists a $n_0 <= n$ and $c > 0$.

This is equivalent to

$$\lim_{n \to \infty} \frac{T(n)}{f(n)} \leq c$$

For example

consider $T(n) = (n+1)^2$.

In this case, we can say that $T(n)$ is $O(n^2)$.

$n^2 + 2n + 1 <= n^2 + 2\,n^2 + n^2 = 4\,n^2$

General Rules for calculating Big-oh

Constant factors don't matter.

For any positive constant *d*, we can say that T(*n*) is *O*(f(*n*)) = *O*(*df*(*n*)).

The reason is that we know that *T*(*n*)<= *c1f*(*n*) for some constant *c1* and all *n*>=n_0.

If we choose *c*=*c1*/*d*, we can see that *T*(*n*)<=c(*df*(*n*)) for *n*>= n_0.

Practically, this means that we would not say "program A is *O*(6*n*)" but rather it has a time complexity of *O*(*n*).

General Rules for calculating Big-oh

Low order terms don't matter.

If the $T(n)$ of program A is some polynomial (i.e. $T(n)=a_k n^k + a_{k-1} n^{k-1} \ldots a_0$) then the approximate running time of A is $O(n^k)$.

This notion can also be applied to any other expression.

Consider $T(n)=2^n+n^3$. As 2^n grows more rapidly than n^3, we can disregard the n^3 term and say that $T(n)$ is $O(2^n)$.

Put more formally, as $\lim_{n\to\infty} \frac{n^3}{2^n} = 0$ we conclude that $T(n)$ is $O(2^n)$.

Every exponential grows faster than a polynomial.

Tightness of Big-oh

While it is technically true to say that

if $T(n) = O(n^2)$ then $T(n) = O(n^3)$ also,

we generally prefer a tight bound on big-oh.

Of course, we must be able to prove it. If we cannot, we must go with the weaker statement.

Big-oh Informal Name

$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	$n \log n$
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(2^n)$	exponential
$O(n!)$	factorial

It should be noted that a sequence of statements (code) that does not involve loops and/or selection statements can be described as $O(1)$.

Operations of Big-oh

$O(f(n))+O(g(n)) = O(\max\{f(n),g(n)\})$

$O(f(n))+O(g(n)) = O(f(n)+g(n))$

$O(f(n))*O(g(n)) = O(f(n)*g(n))$

$O(cf(n)) = O(f(n))$

Some Code Fragment Examples

$O(n)$

```
for(i=0; i<n; i++)
{
        s;
}
```

$O(n^2)$

```
for(i=0;i<n;i++)
{
        for(j=0;j<n;j++)
        {
                s;
        }
}
```

$O(\log_2 n)$

```
h=1;
while(h<=n)
{
        s;
        h = 2*h;
}
```

In this case, the index h jumps (i.e. h takes on values {1,2,4 …}) until *n* is exceeded.

There will be $\log_2 n+1$ iterations, therefore the complexity is $O(\log_2 n)$.

$O(n^2)$

```
for(i=0; i<n; i++)
{
        for(j=0; j<i; j++)
        {
                s;
        }
}
```

As the second loop depends on the first, we can show that the number of steps will be

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

which is ***O*(*n*²).**

Asymptotic Complexity

When $n \to \infty$ (large enough)

- O
- Ω
- Θ
- o
- ω

Asymptotic Complexity

O-notation

A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$,

if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n, i.e.,

if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq c\, g(n) \quad \text{for all } n \geq n_0$$

Asymptotic Complexity

Ω-notation

A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$,

if $t(n)$ is bounded below by some constant multiple of $g(n)$ for all large n, i.e.,

if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \geq c\, g(n) \quad \text{for all } n \geq n_0$$

Asymptotic Complexity

Θ-notation

A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$,

if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n, i.e., if there exist some positive constant c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2\, g(n) \le t(n) \le c_1\, g(n) \quad \text{for all } n \ge n_0$$

Asymptotic Complexity

o *and* ω rarely used.

Using Limits For Comparing Orders of Growth

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} =$$

0 implies that *t(n)* has a smaller order of growth than *g(n)*

C implies that *t(n)* has the same order of growth as *g(n)*

∞ implies that *t(n)* has a greater order of growth than *g(n)*

Features of Asymptotic Complexity

$f(n) = O(g(n)) \approx a \leq b;$

$f(n) = \Omega(g(n)) \approx a \geq b;$

$f(n) = \Theta(g(n)) \approx a = b;$

$f(n) = o(g(n)) \approx a < b;$

$f(n) = \omega(g(n)) \approx a > b.$

Features of Asymptotic Complexity

$f(n)= O(g(n))$, $g(n)= O(h(n)) \Rightarrow f(n)= O(h(n))$;

$f(n)= \Omega(g(n))$, $g(n)= \Omega(h(n)) \Rightarrow f(n)= \Omega(h(n))$;

$f(n)= \Theta(g(n))$, $g(n)= \Theta(h(n)) \Rightarrow f(n)= \Theta(h(n))$;

$f(n)= \Theta(g(n)) \Leftrightarrow g(n)= \Theta(f(n))$.

$f(n)= O(g(n)) \Leftrightarrow g(n)= \Omega(f(n))$;

$f(n)= \Theta(f(n))$;

$f(n)= O(f(n))$;

$f(n)= \Omega(f(n))$.

Analyzing an algorithm

We will use the following criteria:

- Correctness
- Amount of work done
- Amount of space used
- Simplicity
- Optimality

Amount of space used

- The number of memory cells used by a program

1. Prediction of the adequacy of memory space for running a program
2. Allocation of memory for programs in multiuser system
3. Estimation of the size of the problems that can be solved

Amount of space used

- The number of memory cells used by a program

1. Instruction Space
2. Data Space
3. Environment Stack Space

Work space for manipulating the data

Data structure

Amount of space used

The space used by an algorithm is the number of memory cells needed to carry out the computational steps required to solve an instance of the problem excluding the space allocated to hold the input.

All definitions of order of growth and asymptotic bounds pertaining to time complexity carry over to space complexity.

It is clear that the work space can not exceed the running time of an algorithm, as writing into each memory cell requires at least a constant amount of time.

Space Complexity

Thus , if *t(n)* and *s(n)* denote the time and space complexities of an algorithm, then

s(n)=O(t(n))

```
max=a[0];
for (i=1; i<n; i++)
{
   if (a[i]>max)
   max=a[i];
   }
```

Analyzing an algorithm

We will use the following criteria:

- Correctness
- Amount of work done
- Amount of space used
- Simplicity
- Optimality

Simplicity

It is often, though not always, the case that the simplest and most straightforward way of solving a problem is not the most efficient. Yet simplicity in an algorithm is a desirable feature.

It may make verifying the correctness of the algorithm easier, and it makes writing, debugging, and modifying a program for the algorithm easier.

The time needed to produce a debugged program should be considered when choosing an algorithm, but if the program is to be used very often, its efficiency will probably be the determining factor in the choice.

Analyzing an algorithm

We will use the following criteria:

- Correctness
- Amount of work done
- Amount of space used
- Simplicity
- Optimality

Optimality

We say that an algorithm is *optimal* (in the worst case) if there is no algorithm in the class under study that performs fewer basic operations (in the worst case).

How would you know it?

Optimality

Does every algorithm in the class have to be individually analyzed before we can conclude that one is optimal?

No!

We can prove theorems that establish a lower bound on the number of operations needed to solve a problem.

Then any algorithm that performs that number of operations would be optimal.

Optimality

Thus there are two tasks to be carried out in order to find a good algorithm in a class of algorithms being studied, or from another point of view, to answer the theoretical question: How much work is necessary and sufficient to solve the problem?

1. Analyze *A* and find a function *W* such that for inputs of size *n*, *A* does at most *W*(*n*) basic operations.
2. For some function *F*, prove a theorem stating that for any algorithm in the class under consideration, there is some input of size *n* for which the algorithm must perform at least *F*(*n*) basic operations.

If the functions *W* and *F* are equal, then the algorithm *A* is optimal. If not, it may be that there is a better algorithm or that there is a better lower bound.

Example

Problem: Find the largest entry in a list of n numbers.

Upper bound: Suppose the numbers are in an array L. The following algorithm finds the maximum:

Algorithm FINDMAX

Input: L, an array of numbers; $n \geq 1$, the number of entries.

Output: MAX, the largest entry in L.

1. MAX $\leftarrow L(1)$; $i \leftarrow 2$
2. **while** i ≤ n **do**
3. **if** MAX $< L(i)$ **then** MAX $\leftarrow L(i)$
4. $i \leftarrow i + 1$

end while do

Example

Comparisons of list entries are done in line 3, which is executed exactly $n - 1$ times.

Thus $n - 1$ is an upper bound on the number of comparisons necessary to find the maximum in the worst case.

Is there an algorithm that does fewer?

Example

Lower bound:

To establish a lower bound we may assume that the entries in the list are all distinct.

This assumption is permissible because if we can establish a lower bound on worst-case behavior for some subset of inputs (lists with distinct entries), it is a lower bound on worst-case behavior when all valid inputs are considered.

Example

In a list with n distinct entries, $n - 1$ entries are *not* the maximum.

We can conclude that a particular entry is not the maximum only if it is smaller than at least one other entry in the list.

Hence, $n - 1$ entries must be "losers" in comparisons done by the algorithm.

Each comparison has only one loser, so at least $n - 1$ comparisons must be done,

Thus $F(n) = n - 1$ is a lower bound on the number of comparisons needed.

Conclusion: Algorithm **FINDMAX** is optimal.

How ?

Complete Development of an Algo.

1 Statement of the problem
2 Development of a Model
3 Design of the algorithm
4 Correctness of the algorithm
5 Implementation
6 Analysis and complexity of the algorithm
7 Program testing
8 Documentation

Statement of the problem

Before we can understand a problem, We must be able to give it a precise statement. This condition is not, in itself, sufficient for understanding a problem, but it is absolutely necessary.

Developing a precise problem statement is usually a matter of asking the right questions. Some good questions to ask upon encountering a crudely formulated problem are:

Statement of the problem

- Do I understand the vocabulary used in the raw formulation?
- What information has been given?
- What do I want to find out?
- How would I recognize a solution?
- What information is missing, and will any of this information be of use?
- Is any of the given information worthless?
- What assumptions have been made?

Development of a Model

Once a problem has been clearly stated, it is time to formulate it as a mathematical model.

This is a very important step in the overall solution process and it should be given considerable thought.

The choice of a model has substantial influence on the remainder of the solution process.

Development of a Model

There are at lease two basic questions to be asked in setting up a model:

- Which mathematical structures seem best-suited for the problem?
- Are there other problems that have been solved which resemble this one?

Development of a Model

Once a tentative choice of mathematical structure has been made, the problem should then be restated in terms of these mathematical objects. This will be a candidate model if we can give affirmative answers to such questions as:

- Is all the important information in the problem clearly labeled by mathematical objects?
- Is there a mathematical quantity associated with the result sought?
- Have we recognized some useful relations between the objects in the model?
- Can we work with the model? Is it reasonably manipulable?

Design of the algorithm

The choice of a design technique, which is often highly dependent on the choice of model, can greatly influence the effectiveness of a solution algorithm.

Two different algorithms may be correct, but may differ tremendously in their effectiveness.

Correctness of the Algorithm

One of the more difficult, and sometimes more tedious, steps in the development of an algorithm is proving or asserting that the algorithm is correct.

Implementation

Once an algorithm has been stated, say, in terms of a sequence of steps, and one is convinced that it is correct, it is time to implement the algorithm, that is, to code it into a computer program.

- What are the variables?
- What are their types?
- How many arrays, and of what size, are needed?
- Would it be worthwhile to use linked lists?
- What subroutines (possibly already canned) are needed?
- What programming language should be used?

Complete Development of an Algo.

1 Statement of the problem
2 Development of a Model
3 Design of the algorithm
4 Correctness of the algorithm
5 Implementation
6 Analysis and complexity of the algorithm
7 Program testing
8 Documentation

Conventions for Stating Algorithms

Conventions for Stating Algorithms

Algorithm MAX Given N real numbers in a one-dimensional array R(1),R(2),…,R(N), to find M and J such that M=R(J)=max R(K), 1≤k ≤N. In the case where two or more elements of R have the largest value, the value of J remained will be the smallest possible.

Step 0. [Initialize] Set M ← R(1); and J← 1.

Step 1. [N=1?] If N=1 then STOP fi.

Step 2. [Inspect each number] For k←2 to N do step 3 od; and STOP.

Step 3. [Compare] If M<R(K) then set M ← R(K);

and J ← K fi. (M is the largest number in the k^{th} position of the array)

Pseudocode cont.

Algorithm *label-correcting*
Begin
d(s): = 0 and pred(s): = 0
d(j): = ∞ for each j∈N - { s } ;
while some arc(i,j) satisfies $d(j) > d(i)+c_{ij}$ **do**
begin
$d(j):=d(i)+ c_{ij}$
pred(j): = i;
end;
end;

Pseudocode cont.

```
Insertion-Sort (A)
1 for  j ← 2 to length [A]
2       do key ← A[j]
3          ▽Insert A[j] into the sorted sequence A[1...j-1].
4          i ← j-1
5          while i>0 and A[i]>key
6                do  A[i+1]←A[i]
7                    i ← i-1
8          A[i+1] ← key
```

Indentation indicates block structure

End of Section