INTRODUCTION

- Purpose and Contents of this Course: Design and analysis of algorithms

- Definition of Algorithms:
  - A precise statement to solve a problem on a computer
  - A sequence of definite instructions to do a certain job

- Characteristics of Algorithms and Operations:
  - Definiteness of each operation (i.e., clarity)
  - Effectiveness (i.e., doability on a computer)
  - Termination
  - An algorithm has zero or more input and one or more output

- Functions and Procedures:
  - Functions: Algorithms that returns one output
  - Procedures: algorithms that execute a certain job but does not return any output.
    In actuality, procedures can produce a number of outputs as output parameters.

- Design of Algorithms:
  - Devising the algorithm (i.e., method)
  - Expressing the algorithm (computer language)
  - Validating the algorithm (proof of correctness)

- Analysis:
  - Determination of time and space (memory) requirements
• Implementation and Program Testing: Outside the scope

• Devising: Through some algorithmic techniques
  
  – Divide and conquer
  – The greedy method
  – Dynamic programming
  – Graph search methods
  – Backtracking
  – Branch and bound

Expression of Algorithms: (Pseudo language)

• Variable declaration:
  
  integer x, y; real x, y; boolean a, b; char c,d;

  datatype x; (generic)

• Assignment:
  
  X := EXPRESSION; (or X ← EXPRESSION)

  Examples: x ← 1 + 3; y := a*y+2;

• Control structures:
  
  if condition then
  
  a sequence of statments;

  else
  
  a sequence of statements;

  endif

  while condition do
a sequence of statements;

endwhile ;

loop

a sequence of statements;

until condition;

for i=n₁ to n₂ [step d]

a sequence of statements;

endfor

goto Label

Case statement (generalization of if then else ):

Case :

cond₁: stat₁;

cond₂: stat₂;

.  

.  

condₙ: statₙ;

default: stat;

endcase

• Input-Output:

read (X); /*X is a variable or an array*/

print (data) or print (sentence);

• Functions and Procedures:

Function name(parameters)
begin
    Procedure swap(x,y);
        temp := x;
        x := y;
        y := temp;
    end;

begin
    Procedure max(A:integer);
        x := A[1];
        for i = 2 to n do
            if x < A[i] then
                x := A[i];
        end;
        return x;
    end;

begin
    Procedure name(parameters);
        variable declarations;
        body of statements;
        end;

begin
    end

begin
    end

begin
    end

end

Function max(A:integer);
y := temp;
end swap;

**RECURSION**

- A recursive algorithm is an algorithm that calls itself on less input

- Structure of recursive algorithms:

  **Algorithm** $A$(input)
  
  begin
  basis step; /*for minimum size input*/
  call $A$(smaller input); /*recursive step*/
  /*perhaps more recursive calls*/
  combine sub-solutions;
  end ;

- Example:

  **Function** $\text{max}(A(i:j))$
  
  begin
  datatype x,y;
  if $i=j$ then return $(A[i])$; endif ;
  if $j=i+1$ then
  Case :
  default : return $(A[i])$;
  endcase ;
  endif ;
  if $j>i+1$ then
x := \max(A(i:(i+j)/2));
y := \max(A((i+j)/2:j));
if x < y then
  return (y);
else
  return (x);
endif ;
endif ;
end max;

Validation of Algorithms

• Frequently through proof by induction on the input size:
  • Recursion
  • Divide and conquer
  • Greedy method
  • Dynamic programming

Analysis of Algorithms

• What it is: estimation of time and space (memory) requirements

• Why needed:
  • A priori estimation of performance
  • A way for algorithm comparison

• Model:
  • Random access memory (RAM)
  • Arithmetic operations, comparison operations & boolean operations take constant time
• Load and store take constant time

• Time complexity: # of operations as a function of input size

• Space complexity: # of memory words needed by the algorithm

• Example: The non-recursive max: time = (n-1) comparisons, space = 1

Big O Notation

\[ f(n) = O(g(n)) \text{ if } \exists n_0 \text{ and a constant } k \text{ such that } f(n) \leq k \times g(n) \text{ for all } n \geq n_0 \]

\[ f(n) = \Omega(g(n)) \text{ if } \exists n_0 \text{ and a constant } k \text{ such that } f(n) \geq k \times g(n) \text{ for all } n \geq n_0 \]

\[ f(n) = \Theta(g(n)) \text{ if } f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)) \]

**Theorem:** if \( f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0 \), then \( f(n) = O(n^m) \).

**proof:** \[ f(n) \leq |f(n)| \leq |a_m| n^m + \ldots + |a_1| n + |a_0|. \] Therefore,

\[ f(n) \leq (|a_m| + \frac{|a_{m-1}|}{n^1} + \ldots + \frac{|a_1|}{n^{m-1}} + \frac{|a_0|}{n^m}) n^m \leq (|a_m| + \ldots |a_1| + |a_0|) n^m \text{ for all } n. \]

Letting \( k = |a_m| + \ldots + |a_1| + |a_0| \), it follows that \( f(n) \leq kn^m \), and hence \( f(n) = O(n^m) \).

**Method to Compute Time**

• Assignment, single arithmetic and logic operations, comparisons: Constant time

• **if then else**: Time of the body

• **while -for -loop**: If it loops \( n \) times and each iteration takes time \( t \), then the time is \( nt \).
If the \( i \)-th iteration takes \( t_i \), then the time is \( \sum_{i=1}^{n} t_i \).

• Time of the algorithm: sum of the times of the individual statements

**Method to Compute Space**
• Single variables: Constant space

• Arrays (1:n): $n$

• Arrays (1:n,1:m): $n \times m$

• Stacks and queues: maximum size to which the stack/queue grows

Stirling’s Approximation

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$