Pseudo-Code

Idea: Specify *algorithms*, without straightforward programming details.

Idea: Express algorithms with self-explanatory *pseudo-code* that mixes English with syntax that is equivalent to instructions available in most programming languages.

Examples:

$a := b$

Assignment statement that sets $a$ to the value of $b$.

*if* *condition* *then* statement 1 *else* statement 2

If *condition* is true then do statement 1, otherwise do statement 2.
(pseudo-code examples continued)

**for** $i := a$ **to** $b$ **do** statement
Repeatedly set $i := a, i := a+1, i := a+2, \ldots i := b$ and perform the statement.

**for** $i := a$ **to** $b$ **by** $c$ **do** statement
Repeatedly set $i := a, i := a+c, i := a+2c, \ldots$ and perform the statement until $i>b$.

**for** $i := a$ **downto** $b$ **do** statement
Repeatedly set $i := a, i := a−1, i := a−2, \ldots i := b$ and perform the statement.

**for** $1 \leq i \leq n$ **do** statement
Execute the statement for all values of $i$ in this range, the order does not matter; e.g., this is equivalent to: **for** $i := 1$ **to** $n$ **do** statement
(pseudo-code examples continued)

**while condition do** statement
  Repeatedly test *condition* and perform the statement until *condition* is false.

**repeat** statements **until** *condition*
  Repeatedly perform the statements and test *condition* until *condition* is true.

**begin** statement ... statement **end**
  Group these statements together so they can be placed anywhere that you would normally place a single statement.
(pseudo-code examples continued)

function NAME(arguments)
    :
    end

Arguments are evaluated left to right and the result stored in a temporary location so that when a function returns, original values of the arguments are not affected (although an array name can be passed and its contents modified). A function is a **procedure** when it does not return a value (it may still modify global variables or an array whose name was passed as an argument). The statement **return** value immediately exits the function and returns the **value**. For a procedure, **return** exits immediately (without a **return** statement, the procedure returns when its end is reached).

**Array Arguments:** When passing arguments, we sometimes use the expression \( A[i]...A[j] \) as *notation* to mean \( i, j \), and the location in memory of \( A[j] \); so only space for three variables is used to specify this portion of \( A \).
Example: \( n! \) — Pseudo-Code Versus Program Code

Recall that for an integer \( n \geq 1 \), \( n! \) denotes the product of the integers from 1 through \( n \) (define \( n! = 1 \) for \( n \leq 1 \)).

**Pseudo-code for \( n! \):**

\[
\begin{align*}
\text{read } & n \\
& x := 1 \\
\text{for } & i := 2 \text{ to } n \text{ do } x := x \ast i \\
\text{write } & x
\end{align*}
\]

**C Language Code for \( n! \):**

```c
#include <stdio.h>

int main() {
    int x=1, n, i;
    printf("\nEnter n: ");
    scanf("%d",&n);
    for (i=1; i<=n; i++) x = x*i;
    printf("\n%d factorial = %d\n",n,x);
    return(0);
}
```
Idea:

- We would like to be able to explain and develop algorithms at a high level.
- However, we should always understand how pseudo-code corresponds to sequences of instructions.
- In fact, it is important to understand that even typical programming language code is at a much higher level than the basic instructions on a typical computer.
- We want to be able to analyze an algorithm presented by pseudo code with an understanding that there is a direct correspondence with what ultimately takes place on the computer after it is expressed in a programming language and compiled into machine code.
- Actual run time should be proportional to the time derived from the analysis of the pseudo-code.
Example: Assembly code for $n!$ on a single register machine:

($n, x,$ and $i$ are stored in memory locations 1, 2, and 3)

read  read the input $n$ into the accumulator
store 1  store accumulator into memory location 1 (save $n$
load =1  load the constant 1 into the accumulator
store 2  store accumulator into memory location 2 (initialize $x=1$
store 3  store accumulator into memory location 3 (initialize $i=1$

loop:  load 3  load memory location 3 into accumulator (load $i$
subtract 1  subtract memory location 1 from accumulator (subtract $n$
goto($\geq0$) done  go to done if the accumulator is $\geq0$ (done if $i \geq n$
load 3  load memory location 3 into accumulator (load $i$
add =1  add the constant 1 to the accumulator (compute $i+1$
store 3  store accumulator in memory location 3 (save $i$
multiply 2  multiply accumulator by memory location 2 (compute $x*i$
store 2  store accumulator in memory location 2 (save $x$
goto loop  repeat the main loop

done:  load 2  place memory location 2 in accumulator (load $x$
write  write contents of accumulator to output device (write $x$)