Unit 1.1 Foundations

Algorithms

EE/NTHU

Feb. 26, 2018

What is an Algorithm

- In short, algorithm refers to a method that can be used by a computer for the solution of a problem.

Definition 1.1.1. Algorithm

An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

1. Input. Zero of more quantities are externally supplied.
2. Output. At least one quantity is produced.
3. Definiteness. Each instruction is clear and unambiguous.
4. Finiteness. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. Effectiveness. Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in in criterion 3; it also must be feasible.

- Computational procedures have the properties of definiteness and effectiveness.
  - Operating system of a digital computer is an example.

Objectives of Studying Algorithms

- Algorithms can be implemented in different programming languages.
  - A computer program consists of one or more algorithms.
  - An algorithm can also be referred to as a procedure, a function, or a subroutines.
  - Each statement of an algorithm specifies unambiguous operations.
  - Algorithm should be independent to programming languages.
- The objectives of studying algorithms
  1. How to devise algorithms?
  2. How to validate algorithms?
  3. How to analyze algorithms?
  4. How to test a program?
- A good algorithm should be efficient for that specific problem.
  - Efficient in both CPU time and storage space.

Pseudocode Convention

- Algorithms can be implemented in many different programming languages
  - In this class, we use pseudocode to describe algorithms
- Pseudocode is not as rigorous as a programming language
  - Easier to understand by human being but still need to satisfy algorithm's requirements (definiteness, effectiveness)
- The pseudocode adopted is based on `C` language
  - Comments: begin with `//` and continue until the end of a line.
  - Statement:
    - Simple statements followed by `;`
    - Compound statements are grouped within `{` and `}`, also called as a `block`.
  - Identifier convention follows `C`
    - Basic types (`int`, `float`, `char`, etc) are assumed.
    - `struct` (also called `record`) can also be defined.
    - Variables are not declared.
    - Pointers to `struct` variables and their access follow `C` convention.
  - `Assignment`: variable := expression;
  - `Boolean` values: `true` and `false` exist
    - So are logical operators: `and`, `or` and `not`
    - And relational operators: $<$, $\le$, $=$, $\ge$, and $>$.

Loops in the Pseudocode

- Arrays postfixed by $\boxed{[\ ]}$.
  - Two dimensional arrays accessed by $\boxed{A[i,j]}$.
  - Array indexing starts from 1 (Thus, $A[0]$ is usually not defined).

- Loops in the pseudocode are
- while loop

```
while (condition) do {
    statement 1 ;
        ⋮
    statement n ;
}
```

- repeat-until loop

```
repeat {
    statement 1 ;
        ⋮
    statement n;
} until (condition);
```


for Loop

- for loop

```
for variable := value1 to value2 step svalue do {
    statement 1 ;
        ⋮
    statement n ;
}
```

- Note that "step *svalue*" is optional with *svalue* default to $+1$
- The for loop above is equivalent to the while loop below

```
variable := value1;
while ((variable − value2) × svalue ≤ 0) do {
    statement 1 ;
        ⋮
    statement n ;
    variable := variable + svalue;
}
```

- return exits from a function or an algorithm.

Conditional Statements and I/O

- A conditional statement has the following forms:

```
if (condition) then statement ;
if (condition) then statement 1 ; else statement 2 ;
```

- Cascaded-if can be written as

```
switch (variable) {
      case condition 1:   statement 1 ;
            ⋮
      case condition n:   statement n ;
      default:   statement n + 1 ;
}
```

- Input and output of an algorithm are specified by `read` and `write` statements.
  - No `format` is needed for either statement.
- An `error` function is included to handle exception cases (error handling).

Algorithm Declaration

- An `algorithm` consists of a heading and a body. The heading has the form:

```
Algorithm Name(parameter list)
```

- Name is the name of the algorithm and *parameter list* is all the parameters.
  - Simple variables to the algorithm are passed by value or reference.
  - Arrays and structures are passed by reference.
- Body of the algorithm has one or more statements enclosed by { and }.
- A pseudocode example

Algorithm 1.1.2. Max

```
Algorithm Max(A, n)
// Find the largest element of an n-element array A.
{
      Result := A[1] ; // Initialize Result.
      for i := 2 to n do // Loop though all elements.
            if (A[i] > Result) then Result := A[i] ; // Record the larger one.
      return Result; // Done.
}
```

Algorithm Example, Selection Sort

- Sorting problem as an example.
- Input: An array of n elements, $A[1:n]$.
- Problem description: Arrange the elements in increasing (or decreasing) order.
- Solution: From those elements that are currently unsorted, find the smallest one and place it next in the sorted list.

Algorithm 1.1.3. Selection Sort.

```
1 Algorithm SelectionSort(A, n)
2 // Sort the array A[1 : n] into nondecreasing order.
3 {
4       for i := 1 to n do { // for every A[i]
5            j := i; // Initialize j to i
6            for k := i + 1 to n do // Search for the smallest among the rest.
7                 if (A[k] < A[j]) then j := k; // Found, remember it in j.
8            t := A[i]; A[i] := A[j]; A[j] := t; // Swap A[i] and A[j].
9       }
10 }
```


Selection Sort — Correctness

Theorem 1.1.4.

Algorithm SelectionSort(A, n) correctly sorts a set of $n \geq 1$ elements; the result remains in $A[1:n]$ such that $A[1] \leq A[2] \leq \cdots \leq A[n]$.

Proof. For any i, $1 \leq i \leq n$, lines 5-8 select the smallest element among $A[i:n]$ and place it to $A[i]$, thus, $A[i] < A[j]$ for $j > i$.

In addition, these operations does not affect $A[1:i-1]$. Thus, when $i = n$ the entire A is arranged in the non-decreasing order. $\square$

- Note that the upper limit of the for loop in line 4 can be changed to $n-1$ without effecting the correctness of the algorithm.
- The two examples above are both brute-force approach algorithms.
  - Algorithm derived from the definition of the problem.
  - You should be able to write this kind of algorithm with ease.

Recursive Algorithms

- A recursive function is a function that is defined in terms of itself.
- An algorithm is said to be recursive if the algorithm is invoked in the body of the algorithm.
  - An algorithm that calls itself is direct recursive.
  - An algorithm $\mathcal{A}$ is said to be indirect recursive if it calls another function which in turns calls $\mathcal{A}$.
- Using recursion, computer algorithm can be developed quickly.
- Example of recursive function:
  - Factorial function can be defined in mathematical form as

$$\begin{aligned} n! &= 1, && \text{if } n = 1, \\ &= n \times (n-1)!. \end{aligned}$$

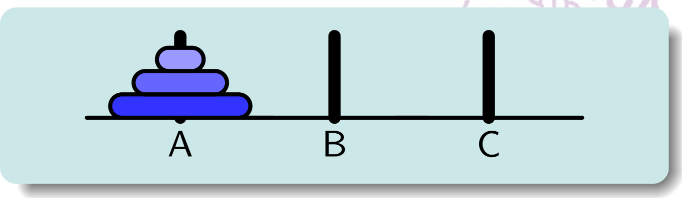
- Then the brute-force approach implementation:

Algorithm 1.1.5. Factorial.

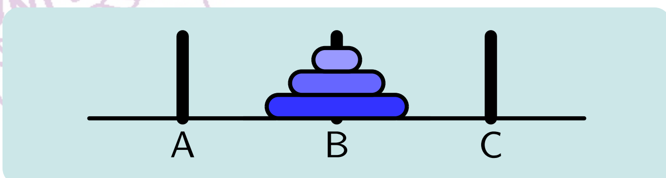
```
Algorithm Factorial(n)
// Calculate factorial function, n!, n ≥ 1.
{
    if (n = 1) return 1; // Termination check.
    return n × Factorial(n − 1); // Recursion formula.
}
```


Tower of Hanoi

- The Tower of Hanoi consists of three rods and n disks of different radius, which can slide onto any rod. All disks are placed in a one stack in ascending order of size on one rod, the smallest at the top, originally. This entire stack is to move to another rod obeying the following rules:
  1. Only one disk can be moved at a time.
  2. Only the top disk of any stack can be moved onto another stack and placed at the top.
  3. No disk can be placed onto a smaller disk.
- Example of 3-disk Tower of Hanoi

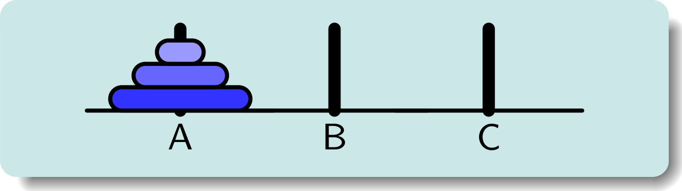


Initial condition.

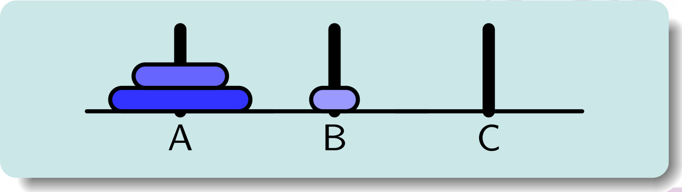


Final state.

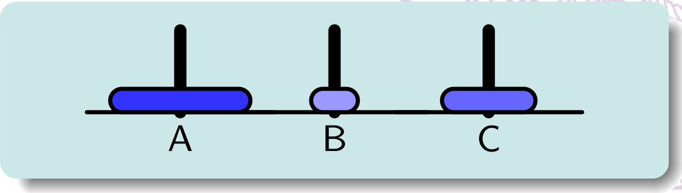
Tower of Hanoi – Solution



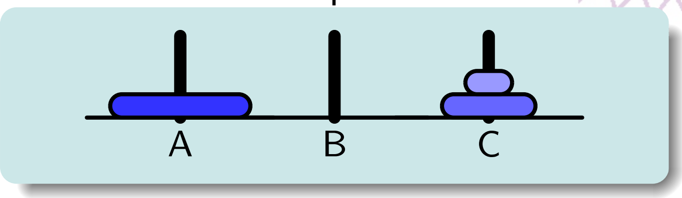
Initial condition.



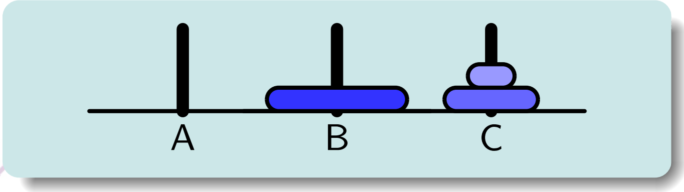
Step 1.



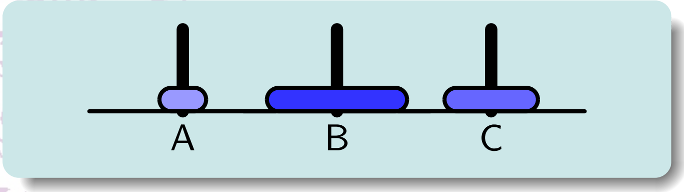
Step 2.



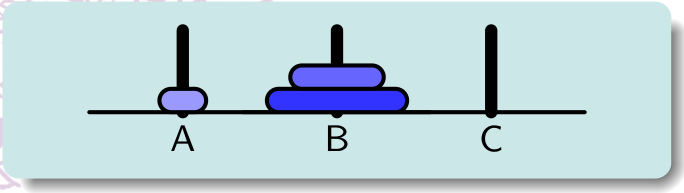
Step 3.



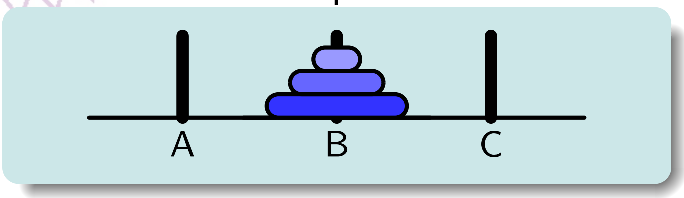
Step 4.



Step 5.



Step 6.



Step 7.

Tower of Hanoi – Algorithm

- Using recursive function Tower of Hanoi problem can be solved easily.
- Assuming n disks to be moved.
- x, y, and z are three rods.

Algorithm 1.1.6. Tower of Hanoi.

```
1 Algorithm TowerOfHanoi(n, x, y, z)
2 // Move the top n disks from rod x to rod y using rod z.
3 {
4       if (n ≥ 1) then { // If there are disks to be moved.
5           TowerOfHanoi(n − 1, x, z, y) ; // move n − 1 disks from x to z using y.
6           write (" Move disk ", n, " from rod ", x, " to rod ", y) ;
7           TowerOfHanoi(n − 1, z, y, x) ; // move n − 1 disks from z to y using x.
8       }
9 }
```

- The algorithm description is simple, the execution can be lengthy.
- For the 3-disk case, as shown in the preceding figure,
  - At the end of line 5, disks are shown as Step 3,
  - Step 4 corresponds to line 6,
  - And line 7 calls itself recursively to reach Step 7.

Tower of Hanoi – Analysis

- How many times the function `TowerOfHanoi` needs to be executed?
  - Let the disks be numbered from 1 to n. Disk n is the largest disk.
  - Disk n needs to be moved only once.
  - But in order to move disk n disk $n-1$ needs to be moved twice.
  - Thus, disk $n-2$ needs to be moved four times.
  - The total number of movements for n-disk problem is

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1. \tag{1.1.1}$$

- The legend has it that when 64-disk Tower of Hanoi is solved, the world would end.
  - Do we need to worry this problem?

Permutations

- Given a set, A, of n elements, it is known that there are $n!$ permutations.
- For example, given the set $\{1, 2, 3\}$ all possible permutations are:
  - $\langle 1,2,3\rangle$, $\langle 1,3,2\rangle$, $\langle 2,1,3\rangle$, $\langle 2,3,1\rangle$, $\langle 3,1,2\rangle$, $\langle 3,2,1\rangle$.
- Using recursive function, all permutation can be generated easily.
- A: set to be permuted, n: number of elements, k: recursion index.

Algorithm 1.1.7. Permutation.

```
Algorithm Permutation(A, k, n)
// To generate all permutation list.
{
    if (k = n) then write (A[1 : n]) ; // output one permutation.
    else // A[k : n] has more permutation, generate them recursively.
        for i := k to n do {
            t := A[k] ; A[k] := A[i] ; A[i] := t; // Swap A[i] with A[k].
            Permutation(A, k + 1, n) ; // All permutations of a[k + 1, n]
            t := A[k] ; A[k] := A[i] ; A[i] := t; // Swap back A[i] and A[k].
        }
}
```

- A call of `Permutation`$(A, 1, n)$ will generate all permutations.

Recursive Algorithms

- As shown by the preceding examples, recursive algorithms are powerful and elegant.
  - Recursive algorithms tend to be short in coding.
- But, recursive algorithms need to use a large program stack space to keep all local variables, in addition to the function arguments.
  - Thus, recursive algorithms may not be the most efficient one in terms of execution time and space.
- To avoid infinite recursion, a recursive algorithm must have termination checks.
  - Line 4 of algorithms (1.1.5), (1.1.6), and (1.1.7).

Summary

- What is an algorithm?
- Objectives of studying algorithms.
- Pseudo conventions.
- Brute force approach
  - Selection sort
  - Proof of correctness
- Recursive algorithms
  - Factorial function
  - Tower of Hanoi
  - Permutations