Unit 1.1 Foundations

Algorithms

EE/NTHU

Feb. 24, 2021

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: $<$, $\leq$, $=$, $\geq$, 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 (*condition* is a boolean expression)

```
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 $\boxed{\texttt{read}}$ and $\boxed{\texttt{write}}$ statements.
  - No `format` is needed for either statement.
- An $\boxed{\texttt{error}}$ 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

```
// Find the largest element in A[1 : n].
// Input: A[1 : n], int n
// Output: max A[i], 1 ≤ i ≤ n
Algorithm Max(A, n)
{
    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.
  - To sort an array $A[1:n]$ into nondecreasing order.
  - Approach: 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.

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


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 4-7 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 $A[1:i-1]$, which is already arranged in nondecreasing order with value less than or equal to $A[i]$. Thus, when $i = n$ the entire A is arranged in the nondecreasing order. □

- Note that the upper limit of the for loop in line 3 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 $\mathcal{A}$.
- A recursive function operates a finite set of objects and has the following 3 elements.
  1. Same operation for the set (and reduced set).
  2. It needs to terminate in finite steps, thus, the successive function calls should reduce the size of the set.
  3. To avoid going into infinite loop, a recursive function needs a termination condition.
- Using recursion, computer algorithm can be developed quickly.

Recursive Algorithm Example – `factorial` function

- Example of recursive function:
  - Factorial function can be defined in mathematical form as

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

- Then the brute-force approach implementation:

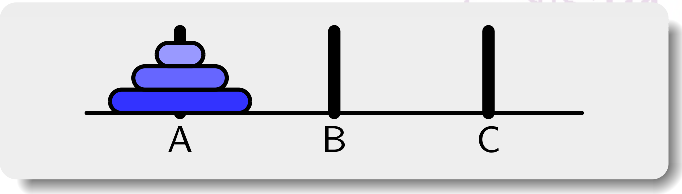
Algorithm 1.1.5. Factorial.

```
  // Calculate n!.
  // Input: int n ≥ 1
  // Output: n!.
1 Algorithm Factorial(n)
2 {
3     if (n = 1) then return 1;  // Termination check.
4     return n × Factorial(n − 1);  // Recursion formula.
5 }
```

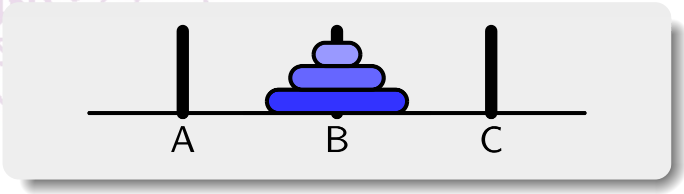
- Note that
  - Same operation – multiplication with result of reduced set.
  - line 3: termination condition,
  - line 4: size reduction for the next recursive call.

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

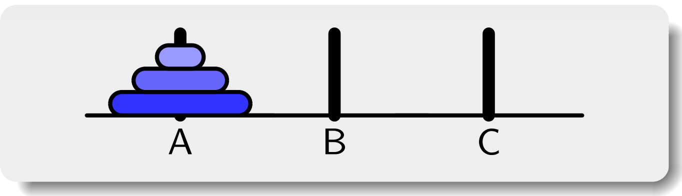


Initial condition.

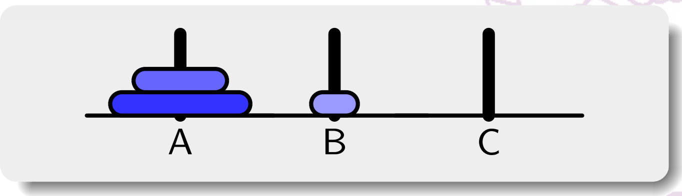


Final state.

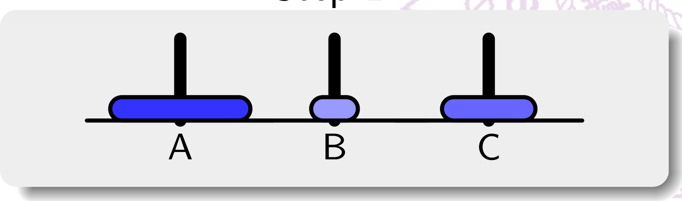
Tower of Hanoi – Solution



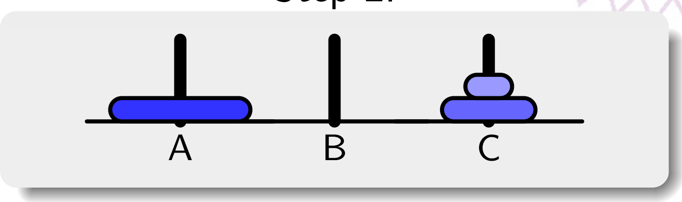
Initial condition.



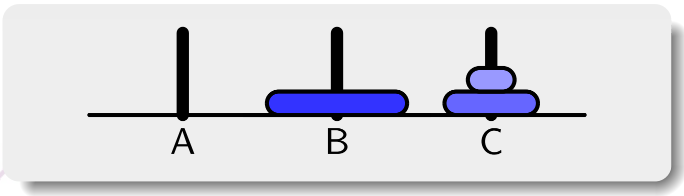
Step 1.



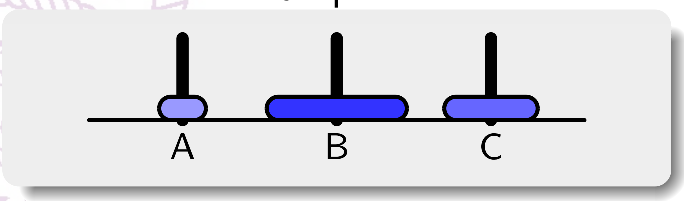
Step 2.



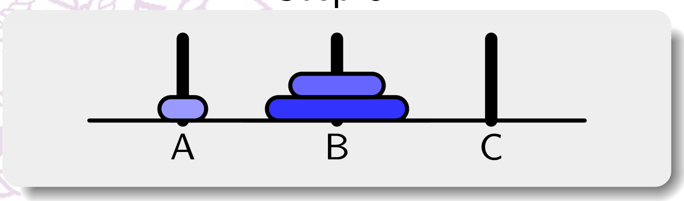
Step 3.



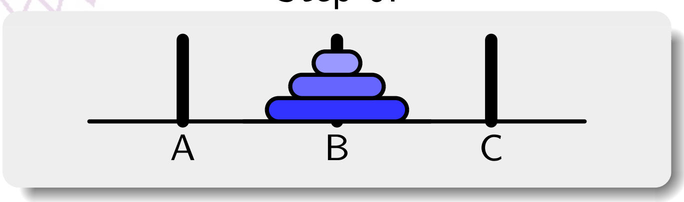
Step 4.



Step 5.



Step 6.



Step 7.

Tower of Hanoi – Algorithm

- The solution, move sequence, shown in the preceding page, is complicated to code.
- 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.

```
// Move n disks from rod x to rod y using rod z.
// Input: n disks; rods: x, y, z
// Output: Legal move sequence.
Algorithm TowerOfHanoi(n, x, y, z)
{
    if (n ≥ 1) then {// If there are disks to be moved.
        TowerOfHanoi(n − 1, x, z, y) ; // move n − 1 disks from x to z using y.
        write (" Move disk", n, " from rod", x, " to rod", y) ;
        TowerOfHanoi(n − 1, z, y, x) ; // move n − 1 disks from z to y using x.
    }
}
```


Tower of Hanoi – Execution

- Execution sequence of moving 3 disks.
- TowerOfHanoi$(3, x, y, z)$ called from main function

```
TowerOfHanoi(2, x, z, y);
    TowerOfHanoi(1, x, y, z);
        TowerOfHanoi(0, x, z, y);
        Move disk 1 from rod x to y;
        TowerOfHanoi(0, z, y, x);
    Move disk 2 from rod x to z;
    TowerOfHanoi(1, y, z, x);
        TowerOfHanoi(0, y, x, z);
        Move disk 1 from rod y to z;
        TowerOfHanoi(0, x, z, y);
Move disk 3 from rod x to y;
TowerOfHanoi(2, z, y, x);
    TowerOfHanoi(1, z, x, y);
        TowerOfHanoi(0, z, y, x);
        Move disk 1 from rod z to x;
        TowerOfHanoi(0, y, x, z);
    Move disk 2 from rod z to y;
    TowerOfHanoi(1, x, y, z);
        TowerOfHanoi(0, x, z, y);
        Move disk 1 from rod x to y;
        TowerOfHanoi(0, z, y, x);
```

- Total number of TowerOfHanoi calls is 15.
  - Can be reduced to 7.
- Disks moved 7 times.
- Recursion depth is 4.
  - Maximum number of copies of TowerOfHanoi function residing in the memory.

Tower of Hanoi – Description

- For the 3-disk case, as shown in the preceding figure,
  - At the end of line 4, disks are shown as Step 3,
  - Step 4 corresponds to line 5,
  - And line 6 calls itself recursively to reach Step 7.
- Note the elements of recursion
  1. Same operation: to move bottom disk from x and y after removing the reduced set,
  2. Size reduction: must move $n-1$ disks to z first, and then move them to y after disk n is in place,
  3. Termination condition: $n = 0$, no disk to move, no recursive call.
- To prove the correctness of Algorithm 1.1.6 note that
  1. Only one disk is moved in line 5.
  2. Only top disk is moved in line 5 since all smaller disks have been moved to rod z in line 4.
  3. No disk is placed onto a smaller disk, since all smaller disks are moved to rod z.
  4. At the end of the algorithm, line 6, entire stack is moved to rod y.
- It can also be proved using induction.

Tower of Hanoi – Analysis

- The algorithm description is simple, the execution can be lengthy.
- 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

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1. \tag{1.1}$$

- 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 distinct elements, then 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.

Algorithm 1.1.7. Permutation.

// Given array $A[1:n]$ of distinct elements, generate all permutations.
// **Input:** $A[1:n]$, int $n > 0$
// **Output:** All permutations of A.
1 **Algorithm** Permutation(A, k, n)
2 {
3   **if** $(k = n)$ **then write** $(A[1:n])$; // output one permutation.
4   **else** // $A[k:n]$ has more permutation, generate them recursively.
5   **for** $i := k$ **to** n **do** {
6             $t := A[k]$; $A[k] := A[i]$; $A[i] := t$; // Swap $A[i]$ with $A[k]$.
7   Permutation$(A, k+1, n)$; // All permutations of $a[k+1, n]$
8             $t := A[k]$; $A[k] := A[i]$; $A[i] := t$; // Swap back $A[i]$ and $A[k]$.
9   }
10 }

Permutation – Analysis

- A call of Permutation$(A, 1, n)$ will generate all permutations.
- Note that recursion elements
  1. Same operation: swap elements in line 6.
  2. Reduction in size: permute $k+1$ to n subarray, in line 7.
  3. Termination condition in line 3.
- Number of operations for Permutation$(A, 1, n)$
  - The recursion depth is n.
    Permutation$(A, 1, n)$ $\rightarrow$ Permutation$(A, 2, n)$ $\rightarrow \cdots \rightarrow$ Permutation(A, n, n)
  - For Permutation$(A, 1, n)$, the loop on lines 5-9 is executed $(n - k + 1)$ times.
  - Thus, the total number of operation is $n \times (n-1) \times (n-2) \times \cdots \times 1 = n!$.
- Note that none of the swap operation exchange the same elements, thus no repeated permutation is generated.

Loops vs. Recursion

- Simple loops can be viewed as
  - Performing the same operation on a finite set.
  - Each iteration reduces the size of the set.
  - When the condition is met, the loop terminates.
- Thus, simple loops can be easily converted to a recursive function.
- Example

```
// Simple loop to get array sum.
// Input: array A; Output: sum = ∑ A[i]
    sum := 0; // init sum to 0.
    for i := 1 to n do
        sum := sum + A[i];
```

```
// Recursive function to get array sum.
// Input: array A; Output: sum = ∑ A[i]
Algorithm ArraySum(A, n)
{
    if (n = 1) return A[1];
    else return A[n]+ ArraySum(A, n − 1);
}
```

- But a recursive function has a larger execution overhead.

Recursive Algorithm Overhead

- A recursive function call need to store the following information in stack space
  - Function arguments and return address,
  - All local variables.
- If the recursion depth is R, then there are R copies of information stored.
- Thus, a recursive function has a larger execution overhead.

- Keeping the overhead in mind, recursive functions are powerful and elegant.
- Good recursive algorithms tend to
  - Short in coding
  - Easier to understand
- A good tool to solve a large number of problems.

Summary

- What is an algorithm?
- Objectives of studying algorithms
- Pseudocode conventions
- Brute force approach
  - Selection sort
  - Proof of correctness
- Recursive algorithms
  - The first method to develop an algorithm
  - Factorial function
  - Tower of Hanoi
  - Permutations