Programming 1

Lecture 12 – Algorithm Basics

Faculty of Information Technology
Hanoi University

Contents

- Problem solving & algorithm
- Pseudo-code
- Searching
- Sorting

Programmer’s mindset

“The computer can only do what you tell it to do. It just does it faster, without getting bored or exhausted.”

(Horstman, 2009)

Algorithm

- **Definition:** A step-by-step list of instructions that, if followed exactly, will solve the problem under consideration.
- Algorithms are often expressed in pseudocode.
- Computer programs are algorithms which are expressed in a specific programming language.

Good algorithm

- **Unambiguous**
  - Precise instructions for what to do at each step and where to go next
- **Executable**
  - Each step can be carried out in practice
- **Terminating**
  - It will eventually come to an end

Don’t think about implementation yet.
Try to focus on “how you want to solve the problem”

Example of ambiguous algorithm

1. Take some pieces of bread
2. Put some becon on one piece
3. Put some salad on the other piece
4. Put the pieces together

What do we have now?

Example of ambiguous algorithm

1. **Take some pieces of bread**

How many pieces?

Example of ambiguous algorithm

2. **Put some becon on one piece**

Put on one side or both sides?

Example of ambiguous algorithm

4. **Put the pieces together**

This sentence doesn’t specify that the pieces of bread should be put together so that the becon and salad are on the inside.

Pseudo-code

- **Definition**: Pseudo-code is a formally-styled language used to describe algorithms.
- It has the following properties:
  - Unlike source code, pseudo code cannot be compiled or executed.
  - Pseudo code uses common concepts of programming languages such as arrays, conditional statements, loops, functions…
  - There is *no standard* for pseudo code.

Why use Pseudo-code?

- Pseudo-code is cross-language
  - Can be translated into almost any language
- Pseudo-code is human-friendly
  - Doesn’t require technical knowledge
- Pseudo-code is faster to write
  - Faster than coding and drawing flowchart

Good Pseudo-code should be…

- Not specific to any single language
- Unambigous
- Balanced
  - Not too specific, not too general
- Use common notations
  - And be consistent about it

Pseudo-code example

Function: Intersect

Input: Two finite sets A, B

Output: A finite set C such that $C = A \cap B$

```
C ← Ø
If |A| > |B|
    Then Swap(A, B)
End
For every x ∈ A Do
    If x ∈ B
        Then C ← C ∪ {x}
    End
End
Return C
```

Pseudo-code example

Function: ArrayMax(A, k, low, high)

Input: Array **A** of **n** integers (**n** > 0)

Output: Largest integer in **A**

```
max ← A[0]
for i ← 1 to n – 1 do
    if max < A[i] then
        max ← A[i]
    end if
end for
return max
```

Pseudo-code example

Set $S = new\ Array(),\ max = 1,\ class = defaultClass$

For $i = 1 \rightarrow N$ { **Set** $S_i = 0$ }

For $i = 1 \rightarrow N - 1$

For $j = i + 1 \rightarrow N$

If $Score_{R_{i,j}}(m) \div T_{i,j} \geq 1$ **then**

Set $S_i = S_i + 1$

Else

Set $S_j = S_j + 1$

For $i = 1 \rightarrow N$

If $S_i > max$ **then** { Set $class = i,\ max = S_i$ }

Return $(class)$

Pseudo-code example

```
begin BubbleSort(list)
   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for
   return list
end BubbleSort
```

Pseudo-code example

```
function BubbleSort(a[]) {
    for i from 1 to N {
        for j from 0 to N-1 {
            if a[j] > a[j+1] {
                t = a[j]
                a[j] = a[j+1]
                a[j+1] = t
            }
        }
    }
    return a
}
```

How to pseudo-code

- Describing how a value is set or changed:
  - `total cost = price + operating cost`
  - `Multiply the balance value by 1.05`
  - `Remove the first and last character from the word`

How to pseudo-code

- Describing decisions and repetitions:
  - `If total cost 1 < total cost 2`
  - `While the balance is less than $20,000`
  - `For each picture in the sequence`

How to pseudo-code

- Use indentation:

```
For each car:
   operating cost = 10 x annual fuel cost
   total cost = purchase price + operating cost
```

Computer program performance

- How to write computer programs that:
  - is fast, responsive, smooth (no lag)
  - uses little RAM
- Causes of delay (lag)
  - I/O time (disk I/O, memory I/O, screen, keyboard, network)
  - CPU time
- What affects CPU time?
  - Calculation, assignment
- What affects memory usage?
  - Variables, arrays, objects, method calls…

Brute-force search algorithm

- **Purpose:** to find the location of a target value in a list of many values (search space), or *"not found"* if the value does not exist in the list.
- **Method:** test every possible value (one by one) in the search space until a match is found.
- **Performance:** Very slow.
- **Effectiveness:** Guaranteed to find the answer (given enough time)
- **Application:** when the search space is random, password cracking

Brute-force search

```
function search(list[], target) {
    for i from 1 to length(list) {
        if list[i] equals target {
            return i // return location
        }
    }
    return -1 // not found
}
```

Find min

```
function min(list[]) {
    min = list[1]
    for i from 2 to length(list) {
        if list[i] < min {
            min = list[i] // update min
        }
    }
    return min
}
```

Find max

```
function max(list[]) {
    max = list[1]
    for i from 2 to length(list) {
        if list[i] > max {
            max = list[i] // update max
        }
    }
    return max
}
```

Calculate sum

```
function sum(list[]) {
    sum = 0
    for i from 1 to length(list) {
        sum ← sum + list[i]
    }
    return sum
}
```

Calculate average

```
function avg(list[]) {
    sum = 0
    for i from 1 to length(list) {
        sum = sum + list[i]
    }
    return sum / length(list)
}
```

Selection sort algorithm

- **Purpose:** to sort a list of comparable values in ascending order.
- **Method:** take out the minimum value and place it at the end of a new list, then repeat the process until there's no value left in the original list.
- **Performance:** Very slow.
- **Application:** when the number of values in the list is small.
- **Advantages:** easy to understand, requires little memory to run.

```
function selection_sort(list[]) {
    for i from 1 to length(list) - 1 {
        min_pos = i // initialize min's position
        for j = i + 1 to length(list) {
            if list[j] < list[min_pos] {
                min_pos = j // update min's position
            }
        }
        // swap list[min_pos] with list[i]
        temp = list[i]
        list[i] = list[min_pos]
        list[min_pos] = temp
    }
    return list[]
}
```

Selection sort trace

- Given the list:

```
list[] = {53, 65, 57, 19, 91, 89}
```

i	j	min_pos
1	-	1
1	2	1
1	3	1
1	4	4
1	5	4
1	6	4

1st iteration: swap list[4] with list[1]

Selection sort trace

- The list:

```
list[] = {19, 65, 57, 53, 91, 89}
```

i	**j**	**min_pos**
2	-	2
2	3	3
2	4	4
2	5	4
2	6	4

2nd iteration: swap `list[4]` with `list[2]`

Selection sort trace

- The list:

```
list[] = {19, 53, 57, 65, 91, 89}
```

i	j	min_pos
3	-	3
3	4	3
3	5	3
3	6	3

3rd iteration: swap list[3] with list[3]
(no change)

Selection sort trace

- The list:

```
list[] = {19, 53, 57, 65, 91, 89}
```

i	j	min_pos
4	-	4
4	5	4
4	6	4

4th iteration: swap list[4] with list[4]
(no change)

Selection sort trace

- The list:

```
list[] = {19, 53, 57, 65, 91, 89}
```

i	j	min_pos
5	-	5
5	6	6

```
5th iteration: swap list[6] with list[5]
```

Selection sort trace

- The output (sorted) list:

```
list[] = {19, 53, 57, 65, 89, 91}
```

Binary search algorithm

- **Purpose:** to find the location of a target value in a list of many values (search space), or *"not found"* if the value does not exist in the list.
- **Requires:** a sorted list.
- **Method:** consider the middle value. If it is bigger than target value, continue the search in the lower part of the list. If it is smaller than the target value, continue the search in the upper part of the list.
- **Performance:** Very fast.
- **Application:** search in database

```
function bin_search(list[], target, low, high):
    if high < low:
        return -1 // not found
    end if
    mid = (low + high) / 2 // get middle position
    if list[mid] > target:
        // look on the left part
        return bin_search(list[], target, low, mid - 1);
    else if list[mid] < target:
        // look on the right part
        return bin_search(list[], target, mid + 1, high);
    else:
        return mid // match found
    end if
end function
```

Binary search trace

- Given the sorted list:

```
list[] = {5, 19, 31, 42, 47, 50, 53, 57, 64, 65, 89, 91}
```

- Perform the search:

```
bin_search(list[], 42, 1, 12)
```

target	=	42
low	=	1 (first position)
high	=	12 (last position)

Binary search trace

`target = 42`

low	high	mid	return	search range
1	12	6	search(low,mid-1)	{5, 19, 31, 42, 47, **50**, 53, 57, 64, 65, 89, 91}
1	5	3	search(mid+1,high)	{5, 19, **31**, 42, 47, 50, 53, 57, 64, 65, 89, 91}
4	5	4	mid	{5, 19, 31, **42**, 47, 50, 53, 57, 64, 65, 89, 91}

search finished in 3 steps

- The search range is halved in each step.
- It takes around $\log_2(N)$ steps for searching in a list of **N** elements.