Algorithms

August 1, 2024

1 Algorithms

Algorithms is the foundation of every program right before they are coded and executed by the computer. By definition, an algorithm refers to a complete and finite set of instructions that accomplishes a particular task. Furthermore, they must be composed of the following:

1. Input: An algorithm may contain zero or more inputs.
2. Output: An algorithm will always output some quantity, regardless of the input supplied.
3. Clarity: An algorithm's each and every instruction must be clear and concise to both human and computer. Furthermore, it should be understandable to the point that a person could write a basic information about the instructions through pen and paper without compromising is effectivity at all costs.
4. Finiteness: An algorithm should always terminate at some point.

Items 1 and 2 implies that for every input (or none at all) an algorithm receives, then it must always produce at least one output. So how do we produce an output? It is through the algorithm's instructions, or otherwise, the process of how the algorithm utilizes the input and transform it into some understandable and definite output. Based on item 3, it must be easily understandable in both human and computer alike.

For example, add 6 or 7 to some x is not a clear instruction since we cannot ascertain which value should we add to the variable x. Other than that, the instruction: compute 1/0 also violates clarity because we are unsure (including the computer) of what the result is.

Lastly, algorithms must end at some point of the program execution. For example, if an algorithm is tasked to sort numbers in ascending order, then it should be able to sort each number to its correct position and then stop its execution once every number has been sorted. However, what if there are too many numbers, like 100,000 of them? Although the algorithm (refer to the Python program below) might be able to sort them accordingly, it might take a while especially when it is exposed to a lot of elements at once.

As such, as part of this course's learning outcomes, you, the student, must be concerned in determining the efficiency of your algorithm such that it will not hamper its instruction execution even when it is exposed to a large amount of inputs (or outputs) at once.

[ ]:
```python
def sortMyList(l : list) -> list:

    # Sort a list of numbers in ascending order
    for x in range(0, len(l) - 1):
        # Assume index at l[x] is least value of the entire list
        swap_idx = x
        for y in range(x + 1, len(l)):
            # If swap_val is greater than the value at index y
            if(l[swap_idx] > l[y]):
                # Change value and index
                swap_idx = y

        # Swap the found least value to index x
        l[x], l[swap_idx] = l[swap_idx], l[x]

    return l
```

[ ]:
```
%%time
# Using sort() with small number of elements
import random

small_l = []
# Random size of list
n = random.randint(10, 50)

for i in range(n):
    small_l.append(random.randint(-100, 100))

print(sortMyList(small_l))
```

```
[-87, -74, -73, -57, -43, -37, 10, 16, 25, 28, 31, 41, 41, 47, 56, 79, 97, 100]
CPU times: total: 0 ns
Wall time: 0 ns
```

This part is divided in two steps. First, we created a numeric list with huge number of elements generated randomly using Python's `random` library, and then use the function above to sort that list.

[ ]:
```
# Using sort() with large number of elements
import random

l = []
n = 100000

for i in range(n):
    l.append(random.randint(-100000, 100000))
```

[ ]:
```
%%time
sortMyList(l)
print("Sorted!")
```

```
Sorted!
CPU times: total: 42.7 s
Wall time: 2min 43s
```

See how long that took compared to the first scenario? Note that when we have 50 or less elements, the `sortMyList()` function was quick, but when we have a hundred thousand elements, it will take at least 1 minute or so. In addition, the performance may depend on one's computer hardware, especially the CPU performance and RAM storage.

You can try copying the code above (both the sorting function and the two lists generated) and run it in your own machine to test the truthfulness of this small test.

We will discuss more about algorithm efficiency in the next chapter. For now, we will proceed to the next subtopic which is about pseudocodes.

2 Pseudocodes

Note that the sorting program above seems to be able to solve the sorting problem it needed. However, do we really need a some programming language code in order to devise an algorithm? The answer is a big NO.

Note that an algorithm is merely a procedure with an input, a set of definite and clear instructions, and an output. It doesn't have to be presented nor interpreted in the form of a programming language or code, but sometimes, for better (or worse), in an alternate English (programming) language known as pseudocode.

To understand what a pseudocode is, let us first define some conventions to be used later. Note that the conventions presented is closely related to Python but can still be interpreted similarly on other programming languages like Java.

1. Comments begin with `//` and continue until the end of the line.
2. Blocks (if any), with matching braces: `{` and `}` will indicate that the next line should be indented. There are also cases where these braces are absent and is replaced by a `:` at the end of a line, indicating that the next line should be indented.
3. Variables or identifiers always start with a letter. Due to the dynamicity of Python, variables will not have their data types specified.
4. Assignment of values to variables is done using the following syntax:

$$\langle \textit{variable} \rangle := \langle \textit{expression} \rangle$$

5. Boolean values are either True or False. These values are usually produced using the following logical operators: $<, \leq, =, \neq, \geq,$ and $>$.
6. Indexes of lists (or arrays) are accessed using the [and] symbols. For example, $A[i]$ means accessing the element of array, A, at index i. If it is a multidimensional array, then it is written as $A[i, j]$, which will access row i and column j of array A. Note that array indices will always start at 0.
7. The loops to be tackled will include while loops and for loops. The `while` loop is structured as this:

```
while ⟨condition⟩ do:
    ⟨statement 1⟩
        ⋮
    ⟨statement n⟩
```

As long as the `while` loop's ⟨*condition*⟩ is true, the statements inside the while loop will be executed. When ⟨*condition*⟩ becomes false, the while loop is exited. Note that the value of the while loop's ⟨*condition*⟩ is always evaluated at the top.
For `for` loops, the general form is as follows:

```
for variable := value1 to value2 do:
    ⟨statement 1⟩
        ⋮
    ⟨statement n⟩
```

In this structure, `value1 to value2` usually refers to the `range()` method in Python, and is interpreted slightly different on other programming languages. Moreover, this structure usually refers to the Python `for` loop that uses indexes to traverse over an entire list. There is a different pseudocode on a `for` loop structure should the `for` loop iterate over values directly rather than indices.

```
for ⟨variable⟩ in ⟨array⟩ do:
    ⟨statement 1⟩
        ⋮
    ⟨statement n⟩
```

8. A conditional statement has the following forms:

```
if ⟨condition⟩ then ⟨statement⟩
if ⟨condition⟩ then ⟨statement 1⟩ else ⟨statement 2⟩
```

They may also be structured as following:

```
if ⟨condition⟩ then:
  ⟨statement⟩
if ⟨condition⟩ then:
  ⟨statement 1⟩
else:
  ⟨statement 2⟩
```

9. Input and output are done using read and write respectively. Generally, it is left for the student to specify the size of input and output of quantities.
10. Functions are declared by a preceding *function* keyword. A function's heading is usually in the following form:

   function Name(⟨parameter list⟩) :

   where `Name` is the name of the function, and (⟨parameter list⟩) refers to the function's parameters (if there are any). The body has one or more statements, usually indented away from the function's column.

As an example, the function below finds the least valued element of an array/list for the first n elements.

```
function Min($A, n$) :
//$A$ is an array of size $m$
    $res := A[1]$
    for $i := 2$ to $n$ do:
        if $A[i] < res$ then $res := A[i]$
    return $res$
```

In the given function above, A and n are the function's parameters. res and i are the function's local variables.

2.1 Translating Programming Problems to Pseudocode

It is very important for you, the student, to be able to logically solve a programming problem first before implementing the code especially as a beginner. Later in this course, (and in your BSIT program), there are complex programming problems that would require hours of thinking before you would be able to come up with a viable solution, which is generally normal. In order for you to come up with this viable solution, it is imperative that your critical thinking skills has to be developed so that you will not have a hard time later on.

2.1.1 Example 1.1: Selection Sort

Suppose that we are given a list composed of integer values and we are tasked to devise an algorithm that will sort these values in ascending order, given that this list of integers has $n \geq 1$ elements. The solution can be devised through the following statement:

Within the range of the list's unsorted elements,
find the index of the element with the lowest value and append it to the sorted list.

This statement may be enough for others to roughly understand the sorting problem, however, it does not meet the requirements of being an algorithm since it is a bit abstract and doesn't reiterate on what steps should be taken. For example, it does not tell us where the sorted list is and whether we should create two instances of a list just to solve the problem.

To deal with this, there are some assumptions to be made. We assume that the sorted list starts from index 0 up to x, where $x + 1$ denotes the amount of elements sorted properly. Algorithm 1.1 is our first attempt at devising the solution.

Algorithm 1.1. Initial Selection Sort

```
for $i := 1$ to $n$ then:
    // Examine $a[i]$ to $a[n]$ and suppose
    // the smallest element is at $a[j]$
    // Then, we swap the values at $a[i]$ and $a[j]$
```

The above algorithm is still not enough. There are two subtasks remaining:

1. How to find the smallest value in the unsorted list, and
2. Swapping values on indexes i and j.

Item number 2 can be easily solved using the pseudocode:

```
$temp := a[i]$
$a[i] := a[j]$
$a[j] := temp$
```

As for item number 1, we can assume that in every iteration, $a[i]$ is always the minimum value of the unsorted list. Then, we can check $a[i+1], a[i+2], \dots, a[n]$, and whenever a smaller element is found at some index k, we regard $a[k]$ as the new minimum value. Eventually, we do this until $a[n]$ and then we are done for that iteration.

Algorithm 1.2 Selection Sort

```
function SelectionSort($a$) :
// Sort the array $a$ from $a[1:n]$
  for $i := 1$ to $n$ do:
    // Index of the minimum element from $i$ to $n$
    $j := i$
    for $k := i+1$ to $n$ do:
      if $(a[k] < a[j])$ then $j := k$
    $temp := a[i]$
    $a[i] := a[j]$
    $a[j] := temp$
```

2.2 Recursive Algorithms

In relation to your Computer Programming 2 course, recursive algorithms are heavily utilized in this course, especially when we reach Topics 5 and 6 later in the Final Term. As defined, a recursive function is a function defined in terms of itself. Most importantly, a recursive algorithm is utilized on situations where we are given a very complicated problem and we want to simplify it by breaking it into smaller chunks.

2.2.1 Example 1.2. Solving Factorials Recursively

A good example would be the factorial operation. For example:

$$n! = n \times (n-1)!$$

$$n! = n \times (n-1) \times (n-2)!$$

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

If you look carefully, we are slowly breaking down our factorials. How? The value of $n!$ is just n multiplied by the factorial of the value before n, which is $(n-1)!$. We can keep on doing this until $n = 0$, where $0! = 1$.

Now, we are going to implement it into pseudocode. Note that we only end up to 1 which happens when $n = 0$, thus, making it our base case since it is the end of breaking down the factorial into its most simple form.

Algorithm 1.3 Factorial

```
function factorial($n$) :
// Get the value of factorial from 1 to $n$
  if $n = 0$ then:
    // Base Case when $n = 0$
    return 1
  else:
    // Recursive call
    return $n * $ factorial($n - 1$)
```

2.2.2 Example 1.3: Truth Value Permutation

During your previous subject, Discrete Structures, you have been introduced to Truth Tables in which the first n initial variables needs to distribute the *True* or *False* values evenly and in every combination possible.

For example, when $n = 2$, then we have the following:

p	q
T	T
T	F
F	T
F	F

If $n = 3$, then:

p	q	r
T	T	T
T	T	F
T	F	T
T	F	F
F	T	T
F	T	F
F	F	T
F	F	F

As you can see in both tables, the distribution of $True$ and $False$ values was even and it exhibits every possible combination for the three propositional variables. This, too, can be solved using a Recursive algorithm! So how do we solve something that looks impressively challenging? First, let's look at the phrase below:

Given n propositional variables, $\{p_1, p_2, p_3, ...\}$*, all possible combinations of their truth values are:*
(1) *the permutation of the truth value of* p_1 *, and*
(2) *every permutation in the truth values between* p_2 and any succeeding propositional variables.

For example, if $n = 2$, then, all possible combinations are:

1. $p_1 = T$ and every permutation of the truth values in (p_2).
2. $p_1 = F$ and every permutation of the truth values in (p_2).

For more clarity, if $n = 3$, then, all possible combinations are:

1. $p_1 = T$ and every permutation of the truth values in (p_2, p_3).
    - $p_2 = T$ and every permutation of the truth values in (p_3).
    - $p_2 = F$ and every permutation of the truth values in (p_3).
2. $p_1 = F$ and every permutation of (p_2, p_3).
    - $p_2 = T$ and every permutation of the truth values in (p_3).
    - $p_2 = F$ and every permutation of the truth values in (p_3).

3 Test Yourself

1. Devise an algorithm that inputs four integers and outputs them in decreasing order.
2. [Permutation Generator] Given a set of $n \geq 1$ elements, the problem is to print all possible permutations of this set. For example, if the set is $\{a, b, c\}$, then the set of permutations is $(a, b, c), (a, c, b), (b, a, c), (b, c, a), (c, a, b), (c, b, a)$. Write your algorithm recursively.