Real Time Signal Processing Laboratory (EE69014)
Learning Lesson - 1

Computational complexity

In computer science, the **computational complexity** of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the string representing the input. Complexity of an algorithm is commonly expressed using big O notation, which excludes coefficients and lower order terms. For example, if the time required by an algorithm on all inputs of size n is at most $5n^2 + 3n$ the asymptotic computational complexity is $O(n^2)$. In other words, $O(n^2)$ means that if the size of the problem (n) doubles then the algorithm will take four times as many steps to complete.

Complexity classes

There are several complexity classes in computational complexity theory. Some important classes, defined using polynomial time are the following,

- **P**: The complexity class of decision problems that can be solved on a deterministic Turing machine in polynomial time.

Examples:

$O(n)$

$O(n^2 + n)$

- NP: The complexity class of decision problems that can be solved on a non-deterministic Turing machine in polynomial time.

Examples:

$O(e^{2n})$

$O(2^n)$

- ZPP: The complexity class of decision problems that can be solved with zero error on a probabilistic Turing machine in polynomial time.
- RP: The complexity class of decision problems that can be solved with 1-sided error on a probabilistic Turing machine in polynomial time.
- BPP: The complexity class of decision problems that can be solved with 2-sided error on a probabilistic Turing machine in polynomial time.

- BQP: The complexity class of decision problems that can be solved with 2-sided error on a quantum Turing machine in polynomial time.

How to Evaluate

Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, where an elementary operation takes a fixed amount of time to perform. Thus the amount of time taken and the number of elementary operations performed by the algorithm differ by at most a constant factor.

Space complexity

A measure of the amount of space, or memory required by an algorithm to solve a given decision problem. It gives an idea about the way in which the amount of storage space required by an algorithm varies with the size of the problem it is solving. Space complexity is normally expressed as an order of magnitude, e.g. $O(n^2)$ means that if the size of the problem (n) doubles then four times as much working storage will be needed.

How to Evaluate

Space complexity depends on the user's code and also on the programming language. Theoretical evaluation of memory requirement is very difficult for large programs written in high level languages. In MATLAB the memory usage can be found out using the following functions.

The first function, 'monitor_memory_feature.m' operates by using the feature ('memstats') command, which displays memory statistics to the MATLAB command prompt. Using the DIARY function, this data can be parsed and used to programmatically stop a function when memory goes over a given limit. The drawback of using the feature ('memstats') command is that its output cannot be suppressed and it will display to the screen.

To run the 'monitor_memory_feature.m' function please type the following the MATLAB command prompt:

[in_use,free,largest_block] = monitor_memory_feature

The second function, 'monitor_memory_whos.m' operates by using the WHOS command and evaluating it within the 'base' workspace. Each variable's memory usage is summed up and converted into megabytes. This function can be run in the background without displaying data to the MATLAB command prompt. However, the memory usage of the workspace is not the only memory used by MATLAB. Typically, the program starts up using approximately 500 MB of memory.

To run the 'monitor_memory_whos.m' function please type the following the MATLAB command prompt:

```
A = magic(1000);

B = phantom(500);

C = peaks(250);

in_use = monitor_memory_whos
```

1. monitor_memory_feature

```
function [ in_use, free, largest_block  ] = monitor_memory_feature( )

%MONITOR_MEMORY grabs the memory usage from the feature('memstats')

%function and returns the amount (1) in use, (2) free, and (3) the largest

%contiguous block.

memtmp = regexp(evalc('feature(''memstats'')'),'(\w*) MB','match');

memtmp = sscanf([memtmp{:}],'%f MB');

in_use = memtmp(1);

free = memtmp(2);

largest_block = memtmp(10)
```

2. monitor_memory_whos

```
function [ memory_in_use ] = monitor_memory_whos( )
%MONITOR_MEMORY_WHOS uses the WHOS command and evaluates inside the BASE
%workspace and sums up the bytes.  The output is displayed in MB.

mem_elements = evalin('base','whos');
if size(mem_elements,1) > 0

   for i = 1:size(mem_elements,1)
      memory_array(i) = mem_elements(i).bytes;
   end

   memory_in_use = sum(memory_array);
   memory_in_use = memory_in_use/1048576;
else
   memory_in_use = 0;
end
```

Execution complexity

This is the time it takes a program to process a given input.

How to Evaluate

Execution complexity depends on the user's code and also on the programming language. In MATLAB the execution time can be found out using MATLAB profiler. Follow the pdf link for further details.

Profiling for Improving Performance - MATLAB & Simulink - MathWorks India

Searching: Linear Search

Linear search or sequential search is a method for finding a particular value in a list that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.

Pseudo code:

For each item in the list:

if that item has the desired value,

stop the search and return the item's location.

Return

Computational Complexity: $O(n)$ (worst case)

The worst case performance scenario for a linear search is that it needs to loop through the entire collection; either because the item is the last one, or because the item isn't found. In other words, if there are n items in the collection, the worst case scenario to find an item is n iterations. This is known as $O(n)$ using the Big O Notation. The speed of search grows linearly with the number of items within your collection.

Sample Test Cases:

Test Case 1:

Find 6 in {3 1 10 5 6 19 13 5 6 2}

Ans: 5

Test Case 2:

Find 16 in {3 1 10 5 6 19 13 5 6 2}

Ans: 0

Test Case 3:

Find 2 in {3 1 10 5 6 19 13 5 6 2}

Ans: 10.

Sorting

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, bubble sort was analyzed as early as 1956.[1] A fundamental limit of comparison sorting algorithms is that they require line arithmic time – $O(nlog(n))$ – in the worst case, though better performance is possible on real-world data (such as almost-sorted data), and algorithms not based on comparison, such as counting sort, can have better performance. Although many consider sorting a solved problem – asymptotically optimal algorithms have been known since the mid-20th century – useful new algorithms are still being invented.

Simple Sort: Selection Sort

This method selects the smallest element in the array and puts it in proper place in the array. To start with, it would be the first position in the array. Once the first element has been positioned, this process is repeated by considering the remaining elements of the array.

Consider the array

[56 30 37 58 19 95 73 25]

The first pass locates the smallest element as 19 and swaps it with the first element 56 to result in the array

[19 30 37 58 56 95 73 25]

The process is repeated to find the next smallest from remaining elements which is 25. It is swapped with the 2nd element to result in the array

[19 25 37 58 56 95 73 30]

The process is now repeated for remaining elements of the array, till all elements have been put in proper places. Note that for this array of size 8, the process is to be executed 7 times. When 7 elements have been placed in proper position, the 8th element will be automatically in its proper place, as it would be largest element.

Pseudo code:

Array A[i]

```
For i = 1 to n-1
    For j = i+1 to n
        min_value_index = j
        if A[j] < A[min_value_index]
            min_value_index = j;
        End if
    End for
    Swap A[i] and A[min_value_index]
End for
```

Computational Complexity: $O(n^2)$ (worst case and best case)

Total number of basis operations,

$C = \sum_{i=1}^{n-1}((\sum_{j=i+1}^{n} 1) + 1)$

Here the basis operations are single comparison and single swapping. The outer for loop variable i runs from 1 to n-1, while the inner for loop j runs from $i + 1$ to n. So total number of comparisons in worst case = $\sum_{i=1}^{n-1}(\sum_{j=i+1}^{n} 1)$ and there is one swap operation for each run of the outer loop.

$$C = \sum_{i=1}^{n-1}((\sum_{j=i-1}^{n} 1) + 1)$$

$$= \sum_{i=1}^{n-1}((\sum_{j=1}^{n} 1 - \sum_{j=1}^{i} 1) + 1)$$

$$= \frac{n^2}{2} + \frac{n}{2} - 1$$

Therefore, the worst case computational complexity of selection sort algorithm is (n^2) .

Sample Test Cases:

Test Case 1:

{10 9 8 7 6 5 4 3 2 1}

Ans: {1 2 3 4 5 6 7 8 9 10}

Test Case 2:

{3 5 6 2 4 67 2 2 1 5}

Ans: {1 2 2 2 3 4 5 5 6 67}

Test Case 3:

{1 2 3 4 5 6 7 8 9 10}

Ans: {1 2 3 4 5 6 7 8 9 10}

Bubble Sort

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller elements "bubble" to the top of the list.

Pseudo code:

Array $A[i]$ *of length* n

repeat

swapped = false

for $i = 1$ *to* $n - 1$ *inclusive do*

if $A[i-1] > A[i]$ *then*

swap them

swapped = true

end if

end for

until not swapped

Computational Complexity: worst case $O(n^2)$, best case $O(n)$

The number of comparison between elements and the number of exchange between elements determine the computational complexity of Bubble Sort algorithm.

• For an array of size n, in the worst case:

1st passage through the inner loop: $n-1$ comparisons and $n-1$ swaps

• ...

• $(n-1)$st passage through the inner loop: one comparison and one swap.

• All together: $C = ((n-1) + (n-2) + \ldots + 1) = n(n-1)/2$

Therefore worst case computational complexity is $O(n^2)$

Sample Test Cases:

Test Case 1:

{10 9 8 7 6 5 4 3 2 1}

Ans: {1 2 3 4 5 6 7 8 9 10}

Test Case 2:

{3 5 6 2 4 67 2 2 1 5}

Ans: {1 2 2 3 4 5 5 6 67}

Test Case 3:

{1 2 3 4 5 6 7 8 9 10}

Ans: {1 2 3 4 5 6 7 8 9 10}

Discrete Fourier Transform (DFT) Algorithm

In mathematics, the **Discrete Fourier Transform** (**DFT**) converts a finite list of equally spaced samples of a function into the list of coefficients of a finite combination of complex sinusoids, ordered by their frequencies, that has those same sample values. It can be said to convert the sampled function from its original domain (often time or position along a line) to the frequency domain.

Pseudo code:

Input signal: $x(n)$

Output signal: $X(k)$

It is assumed that the length of input and output signals are both N. In practice, they may be different.

$W_N = e^{-j2\pi/N}$

For $k = 0$ *to* $N-1$

$temp = 0;$

For $n = 0$ *to* $N-1$

$temp = temp + x(n)W_N^{nk}$

End for

$X(k) = temp$

End for

Computational Complexity: $O(N^2)$

Straightforward implementation of DFT to compute $X(k)$ for $k = 0, 1, \ldots, N-1$ requires:

- N^2 complex multiplications → $4N^2$ real multiplications
- $N(N-1)$ complex addition → $2N(2N-1)$ real additions

Therefore complexity of DFT is $O(N^2)$

Sample Test Cases:

Test Case 1:

1. Generate samples from signal with frequencies 50Hz and 65Hz. Decide the sampling frequency.
2. Determine DFT of the samples with N = 128point.
3. Plot single-sided amplitude spectrum.

Test Case 2:

Repeat Test Case 1 with N = 256.

Test Case 3:

Repeat Test Case 1 with N = 1024.