**ALGORITHM AND COMPLEXITY ANAYLSIS**

* A record is made up of related field
* A field is made up of related character
* A file is made up of related records

The study of the data structure which form the subject matter in this course includes the following:

1. Logical or Mathematical description of the structure of data
2. Implementation of the structure on a computer (i) using computer Science approach
3. Quantitative Analysis of the structure which determines the amount of memory needed to store the data and the time required to process the data. This is what we refers to as complexity of the algorithm

An algorithm is the **best way to represent the solution of a particular problem** in a very simple and efficient way. If we have an algorithm for a specific problem, then we can implement it in any programming language, meaning that an algorithm is independent of any programming language.

Algorithm Design

An important aspect of algorithm design includes creating an efficient algorithm to solve a problem in an efficient way using minimum time and space. To solve a problem , different approaches can be followed. Some of the algorithm can be **efficient with respect to time consumption whereas some may be efficient with respect to memory consumption** .However, one need to keep in mind that both time consumption and memory usage cannot be optimized simultaneously.

If we require an algorithm to run in a lesser time, we have to invest more on memory and if we require an algorithm to run with lesser memory, more time will be required. In solving any computational problem, adequate attention should be paid to type of the problem so as to come up with appropriate algorithm.

Characteristics of an algorithm

1. Algorithms must have unique names

2.Algorithms must have explicitly defined set of inputs and outputs

3.Algorithms are well ordered within unambiguous operations

4.Algorithms halts in a finite amount of time. Algorithms must not run infinitely, there must be an end.

**Analysis/complexity of algorithm**

Algorithm analysis is an important part of computational complexity theory which provides the theoretical **estimation for the required resources needed for the execution of the algorithm.** The required resources are time and space. Hence, analysis of algorithm is simply the determination of the amount of time and space required for its execution.

Usually, the efficiency or running time is stated as a function **relating the input length to the number of steps taken to execute the algorithm (time complexity).**

**Or relating the input length to the amount of the memory space used (space complexity)**

There is a need to discuss the analysis of algorithm and how to choose a better algorithm for a particular problem as one computational problem can be solved by different algorithms.

By considering an algorithm for specific problem, we can begin to develop ‘**pattern recognition**” so that similar problem can be solved by the same algorithm.

Though the objective of algorithms might be the same but they differ from eachother .e.g. a set of numbers could be sorted with different algorithms **but the number of comparisons performed by one algorithm may not be the same with the other ones even with the same input. Hence, time complexity may differ,**

Again, we may need to separately calculate the memory space required by each algorithm.

Analysis of an algorithm is the process of analyzing the problem solving capability of an algorithm in terms of **time of execution and size of the memory for storage during implementation.**

Note that the main concern of the analysis of the algorithm is to determine the required time and space. This is what complexity of algorithm entails.

**The complexity of an algorithm is the function of f(n) which give the running time and the storage space of the algorithm in terms of the size (n) of the input data.**

Generally, we perform the following types of analysis.

1. Worst Case: The maximum number of steps taken on any instance of size a.
2. Average Case: An average number of steps taken on any instance of size a.

To solve a particular problem, we need to consider time as well as space complexity as the program may run on a system where memory is limited but adequate time is available or vice versa. E,g, if we consider **bubble sort and merge sort**; bubble sort do not require additional memory space but merge sort require . Again, time complexity of bubble sort is higher compared to that of merge sort. So, we may need to apply bubble sort if the program needs to run in an environment where memory is limited.

**Asymptotic behaviour of a function**

**In a theoretical analysis of algorithm, it is common to estimate their complexity using the asymptotic approach i.e to estimate the complexity function for arbitrary large input**.

The asymptotic behaviour of a function f(n) refers to the growth of f(n) as the input n gets large. We ignore the small value of n, since we are usually interested in estimating how slow the program execution will be on large input. **The slower the asymptotic growth rate, the better the algorithm**. For instance, a linear algorithm f(n) = n is always asymptotically better than a quadratic one f(n)=n2 (we will discuss more about this later)

**Big “O” Notation (Growth Rate)**

**O(1) = constant complexity**

**O(n) = linear complexity**

**O(n2) = quadratic complexity**

**O(logn)= logarithm complexity**

**O(nlogn)= n logarithm complexity**

**O(n!) = factorial complexity**

O(n!) O(n2)

**(Time)** O(nlogn)

O(n)

O(1)

O(logn)

Input Size

**Big “O” Notation (Growth Rate)**

The complexity of an algorithm is usually define in terms of order of the magnitude of the number of key operation require to perform a function. It is always denoted by “O”. The following list denote some common orders :

O(1) – order 1 , constant complexity time function

O(n) – order n , linear complexity time function

O(n2) – order n2 , quadratic complexity time function

O(logn) – order log n, logn complexity time function

O(nlogn) – order nlogn ,nlogn complexity time function.It is the product of n and log n

O(n!) – order n!, n! complexity time function.

The graphical illustration above shows how various classes of complexity can be compared with each other. The horizontal axis represent the size of the input i.e the input size e.g number of records to be processed in a search algorithm.

The vertical axis represent the computational effort (time) require for the execution of the algorithm in each of the classes. Looking at the diagram carefully, It is an indication of how the time of execution varies with an increase in the size of the input.

**Order O(1)**

Constant time O(1) actually means that an algorithm takes constants time to run, in other words performance is not affected by the size of the input. It can be observed in the diagram above that, at a particular value of time, when there is an increase in input to any length, the time of execution is always constant. Taking any value of (n) the value of (T) is constant. The simplest example is an algorithm **to access the main memory in a computer system**. Also, locating an element in an array generally takes the same amount of time regardless of size. Another important characteristics of O(1) is that there is no guarantee that the algorithm will be very fast but we are only sure that the time taken will always be the same.

**Order O(n)**

An algorithm runs in O(n) if the number of operation require i.e. the time for execution is directly proportional to the item being processed. we can see from the figure that the line of O(n) continues upward but the slope of the line remains the same. One example of this is described as the “**service rate** in queue system”. In this case, it is assumed that the same amount of time is used to service each of the customer on queue. For instance, if the service rate is 5 minute and we have about 10 customers on queue then the total time use to service the customer will be (5x10) 50 minutes

The important point in this is that no matter how many customers on a queue, the time to processed each remain the same. Therefore, we can say that processing time is directly proportional to number of customers. Therefore, is of order O(n)

**Quadratic time O(n2)**

Assuming in a social club made up of six members, the tradition is for each of the member to shake all the remaining member i.e each person in the group must greet and shake hands with every other once. Since there were six people in the group, there will be a total of 5+4+3+2+1 hand shakes = 15 shakes

Assuming we have seven members in the club, then we are going to have 6+5+4+3+2+1 = 21 hand shake and so on.

If we represent the number of individual (members) in the club by (n), the number of hand shakes can be model by **( n2 – n)/2** . Going by the principle or characteristics of O, we should do away with constants hence

(n2 - n )/2 = n2 - n

As the value of n continue to increase or become larger, subtracting from will have less and less overall effects. This means that we can safely ignore the subtraction, leaving us with a complexity of O(

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | % difference |
| 1 | 1 | 0 | 100% |
| 10 | 100 | 90 | 10% |
| 100 | 10,000 | 9,900 | 1.0% |
| 1,000 | 1,000,000 | 999,000 | 0.1% |
| 10,0000 | 100,000,000 | 99,990,000 | 0.01% |

Note that O( is highly applicable where the value of n is small. We should note again **when the value of n is extremely high, the value of the function is high,** meaning that the algorithm will run into completion indefinitely within an indefinite time. **This is applicable in bubble sort algorithm**.

**O(Logn)**

Looking at the diagram, it is obvious that it is the best out of others. Therefore, it is obviously better than O(n) but we must emphasize the fact that it may not be as good as O(1) depending on the nature of the computation. We find this **in binary search algorithm**.

**O(nlogn)**

We can see from the diagram that O(nlogn) is still better than O( but not as good as O(n)

**O(n!)**

From the diagram, it appears that O(n!) is the worst so far, once a very small increment in the value of n will produce an infinite time of execution. Note that the factorial of an integer is a product of itself and all the integer below it.

It is unusual to find a real life example that follows O(n!) and

**Linear Search Algorithm**

Let us look at linear search algorithm. A linear array with N elements and contained an element ITEM to be searched for.

This algorithm find the location LOC of ITEM in array DATA or set LOC=0

1. set K = 1 and LOC=0
2. Repeat step 3(iii) and 4(iv) while LOC=0 and KN
3. If ITEM = DATA then set LOC=K
4. End of step 2(ii) loop

If LOC=0 then

Write: ITEM is not in the array DATA Else

Write : LOC is the location of ITEM

End of If stucture

1. Exit

**Complexity of the linear search algorithm**

The complexity is given by the number of Comparism between ITEM and DATA (K). We have a worst case and an average case.

**The worst case** is when the element is not in the list .

The average case is when the ITEM does appear in DATA and it is likely to occur at any position in the array.

Accordingly, the number of Comparism can be any of the number 1, 2, 3…n and each number occurs with probability

**Binary Search Algorithm**

DATA: 11,22,30,33,40,44,55,60,66,77,80,88,99

To search for 40

BEG=1

END=13

INT(BEG+END)/2

INT( = 14/2 = 7, INT=7 (55)

Since DATA (MID) = 55

Since 40< 55 than

END = MID – 1 = 6. Hence

MID = INT(1+6)/2=3 and so

DATA(MID)=30

Since 40>30, BEG = MID+1=4, Hence

MID= INT(4+6)/2 = 5 and so DATA(MID) = 40

We have found ITEM in location LOC = MID =5

Search successful

**Binary search algorithm works as follows**:

During each stage of our algorithm, the search for ITEM is reduced to a segment element of DATA

DATA:

DATA(BEG), DATA (BEG+1), DATA(BEG+2) … DATA(END)

Note that the variable BEG and END denote respectively, the beginning and the end location of the segment under consideration. The algorithm compares ITEM with the middle element DATA(MID) of the segment where MID is obtain by MID=INT (BEG +END)/2

If DATA (MID) = ITEM, then the search is successful and we set LOC=MID. Otherwise a new segment of data is obtain as follows.

1. If ITEM < DATA (MID) then item can appear only in the left half of the segment i.e DATA (BEG), DATA (BEG+1)…. DATA(MID-1), so we reset, END=MID-1 and begin searching again
2. If ITEM > DATA (MID), then the ITEM can appear only in the right half i.e DATA (MID+1), DATA(MID+2)… DATA(END), so we reset BEG=MID+1 and the searching continue
3. If ITEM is not in DATA then END < BEG

**Binary Searching Algorithms**

1. Set BEG = LB, END = UB and MID = INT (BEG + END)/2
2. Repeat step 3 and 4 while BEG END and DATA (MID) ITEM
3. If ITEM < DATA (MID) then set END = MID-1 else set BEG = MID +1 (END OF if structure)
4. Set MID = INT (BEG+END)/2
5. If DATA (MID) = ITEM, then set

LOC=MID, else

Set Loc=NULL

(End OF IF structure)

1. Exit

Sort for ITEM 85 in this list

DATA: 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, 80, 88, 99

**BEG = 1, END = 13**

**INT )**

**INT = )**

**Students should complete this and submit as an assignment.**

**Limitation of algorithm**

1. The list must be sorted
2. You must have equal access to all element in that list

**Complexity of binary search algorithm**

**The** complexity is measured by the number f(n) of comparisons to locate ITEM in DATA where DATA contains n elements. Observe that each comparison reduces the sample size to half of its initial size. Hence, we require at most f(n)comparisons to locate ITEM where

2f(n) > n or equivalently f(n) =[log2n] + 1

i.e the running time time for the worst case is approximately equal to Log2n. We can also show that **running time for an average case is the same as running time for an worst case.**

In term of space complexity, the algorithm is not economical because extra space will always be required to create more sub list before the original list is finally sorted. This algorithm is an example of Divide and Quaker Algorithm

**If there are 1,000,000 items in the list. How many times will it be sorted before an element is located in the list**

Log2n = f(n)

Log 2 1,000,000 = f(n)

Log2 106 = f(n)

6log210 = f(n)

f(n) = = = 19.9

=20

**Bubble Sort Algorithm**

**Principle of operation**

Suppose the list of number A(1), A(2)…A(N) is in the memory, the bubble sort algorithm works as follows :

1. Compare A(1) and A(2) and arrange them so that A(1) < A(2) then
2. Compare A(2) and A(3) and arrange so that A(2) < A(3)
3. Compare A(3) < A(4), continue until you compare A(N-1) and A(N) and arranged them so that A(N-1) < A(N)

Observe that step 1 involve n-1 comparison (during step1, the largest element is bubbled up to the nth position. This is where the algorithm derived its name “bubble sort algorithm”

When step 1 is completed, A(N) will contain the largest element

At the end of step 2, there will be n-2 comparisons and so on

If we have a list:

A(1), A(2), A(3) ……A(N) , to be sorted in ascending order, then the steps involved are as follows:

1. Compare A(1) and A(2)

If A(1) > A(2) interchange

Else retain the position

1. Compare A(2) and A(3)

If A(2) > A(3) interchange

Else retain the position

3 Compare A(N-1) and A(N)

If A(N-1) > A(N) interchange

Else retain the position

Let A be a list of (n) numbers, sorting A refers to the operation of re-arranging the elements of A such that they are in increasing or decreasing order. For example, suppose A is a list comprising of 8, 4, 19, 2, 7, 13, 5, 16 after sorting in ascending order, A now comprises of 2,4,5,7,8,13,16,19

Suppose the following items are stored in array, A: 32, 51, 27, 85, 66, 23, 13, 57. Illustrate how the bubble sort algorithm could be used to sorts the element in DATA

**The Bubble Sort Algorithm**

1. Repeat steps 2 and 3 for K=1 to N-1
2. Set PTR-1 (initialize pass pointer PTR)
3. Repeat while PTR N-K (execute pass)
4. If DATA (PTR) > DATA (PTR+1) then

Interchange DATA (PTR) and DATA (PTR+1)

End OF IF structure

1. Set PTR = PTR+1

(end of inner loop)

1. Exit

**Complexity of Bubble Sort Algorithm**

For the 1st pass there is n-1 comparison

For the second pass there is n-2 comparison

For the 3rd  pass there is n-3 comparison

And so on

Therefore

f(n) = (n-1)+( n-2) + (n-3) +…+1= n() = = O(.

In term of space, there is no extra space required once the algorithm only interchange the position of elements in the list.

**Quick Sort algorithm**

A quick algorithm is described as a sub list algorithm where the original list is being sub-divided into various sub-list, this serves as a great limitation in the usage of this algorithm since it going to occupy more space(s) in the memory compare to any other algorithm. In other words looking at the complexity of the algorithms in terms of space, it is not economical

A is an array with N elements, Parameters BEG and END contain the boundary values of the sub-list of “A” to which this procedure applies, Loc: keeps the position of the first elements A(BEG) of the sub-list during the procedure,

The local LEFT and RIGHT will contain the boundary values of the list of elements that have not been scanned,

1. Initialize

Set LEFT: BEG, RIGHT: END and Loc = BEG

1. Scan from right to left
2. Repeat while A(Loc) A(RIGHT) and Loc RIGHT

Right = RIGHT -1

(End of loop)

1. If Loc = RIGHT, then Return
2. If A(Loc) > A( RIGHT), then
3. Interchange A(Loc) and A( RIGHT)

TEMP = A(Loc), A(Loc) = A(RIGHT)

A(RIGHT) = TEMP

1. Set Loc = RIGHT
2. Go to step 3

(End OF IF structure)

1. Scan from LEFT to RIGHT
2. Repeat while A(LEFT) A(Loc) and LEFT Loc

LEFT = LEFT +1

End of loop

1. If Loc = LEFT, then Return

c A(LEFT) > A(Loc) then

i Interchange A(LEFT) and A(Loc)

TEMP = A(Loc), A(Loc) = A(LEFT)

A(LEFT) = TEMP

ii Set Loc = LEFT

iii Go to step 2

End OF IF Structure

1st 44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66

2nd 22, 33, 11, 55, 77, 90, 40, 60, 99, 44, 88, 66

3rd 22, 33, 11, 44, 77, 90, 40, 60, 99, 55, 88, 66

4th 22, 33, 11, 40, 77, 90, 44, 60, 99, 55, 88, 66

5th 22, 33, 11, 40, 44, 90, 77, 60, 99, 55, 88, 66

**Complexity of the algorithm**

**Worst case:** The worst case occurs when the list is already sorted. The fist element will require n comparison to to recognize that it remains in the first position. The first sublist will be empty., but the second sub list will have n-1 elements. Accordingly,the second sub list will require n-1 comparisons to recognize that it remains in the second position and so on. Consequently, there will be a total of :

F(n) = n+(n-1) + (n-2) + (n-3) + … + 1 = n() = O = O( comparisons.

Observe that this is equivalent to the complexity of bubble sort algorithm.

**Average Case**

The complexity f(n) = O(n log n) is the average case complexity