Analysis of Algorithm

Importance of Analyze Algorithm

-Need to understand limitations of various Algorithms for solving a problem.

-Need to understand relationship between problem size and running time.

-Need to learn how to analyze an Algorithm’s running time without coding it.

-Need to learn technique for writing more efficient code.

The overall goal of Analysis of Algorithm is on understanding of the complexity of Algorithm.

Complexity of Algorithm Is simply the amount of work the algorithm perform to complete its task.

or

A measure of the Analysis of Algorithms or performance of Algorithms.

There are mainly two types of complexity

1.Space complexity

2.Time complexity

<u>Space complexity</u>

of an algorithm is the amount of memory that it needs to run to completion.

The way in which the amount of storage space required by on algorithm varies with the size of the problem it is solving.

The amount of time required by on algorithm varies with the size of the problem.

as the complexity (space & time) grows, the size of the problem also grows

Growth function – Big-Oh (Big-O) notation

As the size grows, the growth rate becomes the most important aspect of the complexity of Algorithm. For this reason, a notation for expressing growth rate is used called Big-O notation.

Big-O notation indicates the growth rate

. Big-O function has a parameter n, where n is usually the size of the input to the algorithm.

. Big-O stands for “order of”

Definitions:-

T(n) = O(f(n)) is read as T of N is equal to Big-O of ‘f’ of n. It means that, growth rate of T(n) is less than or equal to growth rate of f(n).

where

T(n) is a non-negative valued function n be the size of the input.

Rules for Big-O notation

1. Ignore added constants.

Because they become insignificant as the data increases

eg: n+50 --> O(n)

eg: a+n+c --> O(n)

2. Ignore constant multiplies

(same reason)

eg: 10*n --> O(n)

50*n --> O(n)

3. When adding terms together, variable having highest power is selected. Use the larger of two
eg: n+n2 --> n2   O(n2)
    4n+20n4+10-->O(n4)

4. Write multiplication more compactly
    eg: n*n-->n2 O(n2)

5. If there are no loops or function call that can’t form a loop, the complexity is O(1).

6. For a sequence of S1,S2,S3,…Sk statements, running time is maximum of running time of individual statements.

Example

find the T(n) of 3n2+10n+10

3n2+10n+10 becomes=========>3n2+10n

(using rule 1)

3n2+10n becomes======>3n2 (using rule 3)

3n2 becomes =======>n2

(using rule 2)

there fore T(n)= n2

The Most important Notations are:

O(1):-

- An algorithm with this running time is known as constant running time. That is O(1).
- Simple program statement take a constant amount of time.
- This means the algorithm always take the same amount of time regardless of the size of the input.

- For example, an algorithm which performs 7 multiplications has a constant running time. An algorithm which finishes in under a year has a constant running time.
- constant time is the best running time of an algorithm
- some examples are: Inserting an element into stack, array, linked list. or deleting .....

Example 1:-

```
 i=1;
printf(“%d”,i);
printf(“%d”,n);
printf(“%d”,i*n);
printf(“%d”,i*i);
printf(“%d”,n);
printf(“%d”,i*n);
printf(“%d”,i*i);
```

T(n) for each line is 1.

T(n) for the entire line algorithm is

1+1+1+1+1+1+1+1.

Using rule 1, the complexity is O(1).

so T(n) for entire algorithm is O(1).

- If there are 5 statements, we don’t say O(5). we always use O(1) to represent constant time.

- The amount of time spent executing the code will be the same, if there were 0 items or 10000 items.

O(n):

- An algorithm with this running time is known as “linear running time”.
- This is basically means that the amount of time to run the algorithm is proportional to the size of the input.
- Execution time increases linearly with the number of items you have.

- O(n) means that, if the size of the problem doubles, then running time and space are also doubles.
- Some examples are: Searching through an unordered list, increasing every element of an array.

Example

```
For(i=0;i<n;i++)
Printf("%d",i)
```

Here,

The amount of time required to execute the first line=n that is O(n).

The amount of time required to execute the second line=1 that is O(1).

T(n) for the entire 2 line algorithm is 1 x n------->n

So the algorithm is O(n)

T(n)=O(log n)

- An algorithm with this running time is called logarithmic running time.
- This means that, as the size of the input increases by a factor of n, the running time increases by a factor of the logarithm of n.
- For a binary search algorithm, we devide the list in half, in quarters, in eights, etc until we find the key.

- For example , there are 100 elements and it would take no more than 7 executions. Why 7? The answer is that 26<100<27. Written another way, 6<log 100< 7, where log is in base 2

Therefor T(n) for binary search is O(logn)

- The running time is better than O(n), but not as good as O(1).

- Example code:

$T(n)=O(n^2)$

- An algorithm with this running time is known as quadratic running time.
- This means that whenever you increases the size of the input by a factor of n, the running time increases by a factor of $n^{2.}$
- execution time increases with the squire of the number of items you

  have.

- $O(n^2)$ means if the size of the problem doubles then four time as much storage or time will be needed.
- Example:

O(nlogn)

- The running time is better than O(n2), but not as good as O(n)
- The fastest sorting algorithms including merge sort, and quick sort have O(nlogn) running time.

Example:

O(**2^n**)

- An algorithm with this running time is known as exponential.
- This means that, its running time will doubles every time you add another element to the input.
- An algorithm which takes an input with 30 elements need to perform as many as 1 billion steps. If the input has 40 elements, then 1 trillion steps may be necessary.
- Ex: factoring large numbers expressed in binary.

O(n^n)

- This is known as polynomial growth
- Algorithm which takes 10 elements of input may need to perform 10 billion steps.

O(n!)

- An algorithm with this running time is known as factorial.
- If the input size is n, then the total time will be proportional to nx(n-1)x(n-2)x….2x1.
- For example if n=8. The no of steps will be proportional to 8x7x6x5x4x3x2x1=40520
- If the input size reaches 15, the no of steps may exceed 1 trillion.

Three time complexity functions

(or 3 different cases of time complexity)

1) Worst Case
2) Average Case
3) Best Case

```
int linearsearch (int a [ ], int first, int last, int key)
{
        for (int i = first; i < = last; i ++)
        {
                if (key = = a [i])
                {
                return i;          // successfully found the
                }         // key and return location
        }
        return – 1;               // failed to find key element
}
```

Worst Case

The worse case time complexity of the algorithm is the function defined by the maximum number of operations performed, taken across all instances of size n. The worst-case time complexity is an upper bound on the running time for any input.

The worst case occurs when the search term is in the last slot in the array, or is not in the array.

The number of comparisons in worst case = size of the array = n.

Time complexity = O(n)

Avarage Case

- On average, the search term will be somewhere in the middle of the array.

- The average Case takes - **(*n*+1)/2** comparisons.

- Time complexity = O(n)

Best Case

The best case time complexity of the algorithm is the function defined by the minimum number of operations performed, taken across all instances of size n.

•The best case occurs when the search term is in the first slot in the array.

•Number of comparisons in best case = 1.

•Time complexity = O(1).

Algorithm	Complexity
Bubble Sort Insertion Sort	**Worst complexity:** n^2 **Average complexity:** n^2 **Best complexity:** n **Space complexity:** 1
Selection sort	Worst complexity: n^2 Average complexity: n^2 Best complexity: n^2 Space complexity: 1
Merge sort	Worst complexity: n*log(n) Average complexity: n*log(n) Best complexity: n*log(n) Space complexity: n

Algorithm	Complexity
Linear search	Worst complexity: O(n) Average complexity: O(n) Worst-case space complexity: O(1)
Binary search	Worst complexity: O(log n) Average complexity: O(log n) Best complexity: O(1)