Data Structures and Algorithms

Compiled By
Getahun F.

Chapter 2 complexity analysis in algorithm

Algorithm Analysis

- Algorithm is a step by step procedure to solve a problem.
  - E.g. baking cake, industrial activities, Student Registration, etc, all need an algorithm to follow.
- The purpose of an algorithm is to ***accept input values***, ***to change a value*** hold by a data structure, ***to re-organize the data structure*** itself (e.g. sorting), ***to display the content of the data structure***, and so on.
- More than one algorithm is possible for the same task.

Properties of Algorithms

- ***Finiteness:*** any algorithm should have finite number of steps to be followed.
- ***Absence of ambiguity:*** the algorithm should have one and only one interpretation during execution.
- ***Sequential:*** it should have a start step and a halt step for a final result
- ***Feasibility:*** the possibility of each algorithm to be executed.
- ***Input/Output:*** ***takes*** an input, and produce an output.

Algorithm Analysis Concepts

- ***Algorithm analysis*** refers to the process of determining how much computing time and storage that algorithms will require.
- In other words, it's a process of predicting the resource requirement of algorithms in a given environment.
- ***Complexity analysis*** is concerned with determining the efficiency of algorithms.
- There are two things to consider:
  - » **Time Complexity**: Determine the approximate number of operations required to solve a problem of size n.
  - » **Space Complexity:** Determine the approximate memory required to solve a problem of size n.
- There are ***two approaches*** to measure the efficiency of algorithms: ***Computational and Asymptotic.***

Computational (Empirical) Analysis

- The total running time of the program is considered.
- It uses the system time to calculate the running time because running time is usually treated as the most important analysis parameter.
- However, it is difficult to use actual clock-time as a consistent measure of an algorithm's efficiency, because clock-time can vary based on many things.
  - Processor speed
  - Current processor load
  - Input size of the given algorithm and
  - Software environment (multitasking, single tasking,…)

Asymptotic (Theoretical) Analysis

- Determining the quantity of resources required mathematically (execution time, memory space, etc.) needed by each algorithm.
- Consider $t = f(n) = n^2 + 5n$; For all $n>5$, n^2 is largest, and for very large n, the 5n term is insignificant.
- Therefore we can approximate f(n) by the n^2 term only. This is called ***asymptotic complexity***.
- An approximation of the computational complexity that holds for large n.
- Used when it is difficult or unnecessary to determine true computational complexity.
- Usually it is difficult to determine computational complexity. So, asymptotic complexity is the most common measure.

Complexity Analysis...

- Complexity analysis involves ***two distinct phases***:
  - » ***Algorithm analysis*** - Find f(n)- helps to determine the complexity of an algorithm.
  - » ***Order of Magnitude*** - g(n) belongs to O(f(n)) - helps to determine the category of the complexity to which it belongs.
- There is no generally accepted set of rules for algorithm analysis.
- However, an exact count of operations is commonly used.

Analysis Rule

1. Assume an arbitrary time unit
2. Execution of one of the following operations takes time ***1***:
   - Assignment statement E.g. Sum=0;
   - Single I/O statement;. E.g. cin>>sum; cout<<sum;
   - Single Boolean statement. E.g. !done
   - Single arithmetic. E.g. a+b
   - Function return. E.g. return(sum);
3. Selection statement
   - Time for condition evaluation + the maximum time of its clauses
4. Loop statement
   - Σ(no of iteration) +1 + n+1 + n (initialization time + checking + update)
5. For function call
   - 1+ time(parameters) + body time

Algorithm Analysis...

- ***Example 1:*** Calculate T(n) for the following

```
int k=0;
cout<<"Enter an integer";
cin>>n;
for (i=0;i<n;i++)
    k++;
```

- $T(n) = 1+1+1+(1+n+1+n+n)$

  $= 5+3n$

Algorithm Analysis...

- ***Example 2:***

```
int i=0;
 while (i<n){ x++;
  i++;
 }
int j=1; while(j<=10){
  x++;
  j++
 }
```

- T(n) = 1+n+1+n+n+1+11+10+10

    = 3n+34

Algorithm Analysis...

- ***Example 3:***

```
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
        k++;
```

- $T(n) = 1+n+1+n+n(1+n+1+n+n)$

  $= 3n^2+4n+2$

Algorithm Analysis...

- ***Example 4:***

```
int sum=0;
if(test==1){
      for (int i=1;i<=n;i++)
      sum=sum+i;
}
else
{ cout<<sum;}
```

- T(n) = 1+1+Max(1+n+1+n+n+n,1)

  = 4n+4

Exercise

- Calculate T(n) for the following codes
- 1)
```
int sum=0;
for(i=1;i<=n;i++)
 for(j=1;j<=m;j++)
  sum++;
```
- 2)
```
int sum=0;
for(i=1;i<=n;i++)
 for(j=1;j<=i;j++)
sum++;
```

Algorithm Analysis Categories

- Algorithm must be examined under different situations to correctly determine their efficiency for accurate comparisons.
- ***Best Case Analysis:***
  - » Assumes that data is arranged in the most advantageous order.
  - » It also assumes the minimum input size.
- ***For example:***
  - » ***For sorting*** – the best case is if the data is arranged in the required order.
  - » ***For searching*** – the required item is found at the first position.
- ***Note***: Best Case computes the lower boundary of T(n)
- It causes fewest number of executions

Cont…

- ***Worst Case Analysis:***
  - » Assumes that data is arranged in the disadvantageous order.
  - » It also assumes that the input size is infinite.
- ***For example:***
  - » ***For sorting*** – data is arranged in opposite required order.
  - » ***For searching*** – the required item is found at the end of the item or the item is missing.
- It computes the upper bound of T(n) and causes maximum number of execution.

Cont…

- ***Average Case Analysis:***
  - Assumes that data is found in random order.
  - It also assumes random or average input size.
- ***For example:***
  - ***For sorting*** – data is in random order.
  - ***For searching*** – the required item is found at any position or missing.
- It computes optimal bound of T(n).
- It also causes average number of execution.
- Best case and average case can not be used to estimate (determine) complexity of algorithms
- Worst case is the best to determine the complexity of algorithms.

Order of Magnitude

- Order of Magnitude refers to the rate at which the storage or time grows as function of problem size (function (n)).
- It is expressed in terms of its relationship to some known functions – ***asymptotic analysis.***
- Asymptotic complexity of an algorithm is an approximation of the computational complexity that holds for large amounts of input data.
- Types of Asymptotic Notations
  - » Big - O Notation
  - » Big – Omega (Ω)
  - » Big – Theta (Θ)
  - » Little o (small o)
  - » OO Notation

Big – O (Oh) Notation

- ***Definition :*** The function T(n) is O(F(n)) if there exist constants c and N such that $T(n) \le c.F(n)$ for all $n \ge N$.
- As n increases, T(n) grows no faster than F(n) or in the long run (for large n) T grows at most as fast as F.
- It computes the tight upper bound of T(n).
- Describes the worst case analysis.

Cont…

- ***Example 1:***
- Find $F(n)$ such that $T(n) = O(F(n))$ for

  $T(n) = 3n+5$

  » $T(n) = O(n)$

  The complexity o the function increases with the increment of the n value

Cont…

- ***Example 2:***
  - » $T(n) = n^2 + 5n$
  - » $F(n) = n^2$
  - » $T(n) = O(n^2)$
- In the above example, the n^2 term becomes larger than the 5n term at n>5
- The worst case for this type of function is n^2 for thevalue of n >5

Big – Ω (Omega) Notation

- ***Definition:*** The function T(n) is Ω(F(n)) if there exist constants c and N such that T(n) ≥ c.F(n) for all n ≥ N.
- As n increases T(n) grows no slower than F(n) or in the long run (for large n) T grows at least as fast as F.
- It computes the tight lower bound of T(n).
- Describes the best case analysis.

Big – Θ (Theta) Notation

Theta notation describes the average-case scenario of an algorithm's time complexity.

Individual Assignment

- Little – o (small - o) Notation
- Big - OO – Notation
- Amortized complexity

Exercise

- Find ***Big – O*** of the following Algorithms:
- 1)
```
for (i=1;i<=n;i++)
      cout<<i;
```
- 2)
```
for (i=1;i<=n;i++)
       for(j=1;j<=n;j++)
           cout<<i;
```
- 3)
```
for(i=1;i<=n;i++)
   sum=sum+i;
      for (i=1;i<=n;i++)
         for(j=1;j<=m;j++)
             sum++;
```