Unit-1: Foundation of Algorithm Analysis

Algorithms and its properties

- An algorithm is a finite set of computational instructions, each instruction can be executed in finite time, to perform computation or problem solving by giving some value, or set of values as input to produce some value, or set of values as output.
- Algorithms are not dependent on a particular machine, programming language or compilers i.e. algorithms run in same manner everywhere.

An algorithm is a sequence of steps to solve any problem.

- An algorithm can be specified as:
  - Natural language
  - Pseudo code
  - Flow chart
  - Actual program code in a programming language.

Examples of problems

- You are given two numbers, how do you find the Greatest Common Divisor.
- Given an array of numbers, how do you sort them?

We need algorithms to understand the basic concepts of the Computer Science, programming. Where the computations are done and to understand the input output relation of the problem we must be able to understand the steps involved in getting output(s) from the given input(s).

You need designing concepts of the algorithms because if you only study the algorithms then you are bound to those algorithms and selection among the available algorithms. However if you have knowledge about design then you can attempt to improve the performance using different design principles.

The analysis of the algorithms gives a good insight of the algorithms under study. Analysis of algorithms tries to answer few questions like; is the algorithm correct? i.e. the Algorithm generates the required result or not?, does the algorithm terminate for all the inputs under problem domain? The other issues of analysis are efficiency, optimality, etc. So knowing the different aspects of different algorithms on the similar problem domain we can choose the better algorithm for our

need. This can be done by knowing the resources needed for the algorithm for its execution. Two most important resources are the **time and the space.**

Properties of algorithm

An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

Input & output: The algorithm must accept zero or more input and must produce at least one output.

Definiteness: Each step must be clear and unambiguous.

Finiteness: Algorithms must terminate after finite time or steps.

Correctness: Correct set of output values must be produced from the each set of inputs.

Effectiveness: The instructions must be basic to carry out and must be feasible. Each step must be carried out in finite time.

RAM Model

The RAM (Random Access Machine) model is base model for our study of design and analysis of algorithms to have design and analysis in machine independent scenario. This model assumes all algorithms must be implemented in a single processor machine. In this model each basic operations (+, -) takes 1 step, loops and subroutines are not basic operations. Each memory reference is 1 step. We measure run time of algorithm by counting the steps.

The RAM (Random Access Machine) model of computation measures the run time of an algorithm by summing up the number of steps needed to execute the algorithm on a set of data. The RAM model operates by the following principles:

- Basic operations (+, *, =) are considered to be simple operations that take one time step.
- Loops and subroutines are complex operations composed of multiple time steps.
- All memory access takes exactly one time step

Time and Space Complexity

— Same problem can be solved using different algorithms. Among different algorithms some would solve the problem efficiently and some would not.

- The differences between the algorithms may be immaterial for processing a small number of data items, but these differences grow with the amount of data. To compare the efficiency of algorithms, a measure of the degree of difficulty of an algorithm called *computational complexity* was developed.
- **Computational Complexity** is the measure which measures the efficiency of an algorithm while the amount of data to be processed by algorithm grow.
- Computational complexity indicates how much effort is needed to apply an algorithm or how costly it is.
- This cost can be measured in a variety of ways, and the particular context determines its meaning.
- We focus on the two **important efficiency (cost) criteria**: **time** and **space**
- Analyzing the algorithm means predicting the resource requirement to execute any algorithm. The most important resources are **running time and space (memory) required to run the algorithm**. If the computational complexity is concerned with time such complexity measure is known as **time complexity** and if the computational complexity is concerned with space then such complexity is known as **space complexity**.
- **Space complexity** of an algorithm represents the amount of memory space needed the algorithm in its life cycle. In RAM model it is computed by summing the number of steps required.
- **Time Complexity** of an algorithm is the representation of the amount of time required by the algorithm to execute to completion. . In RAM model it is computed by summing the number of memory references.

Mathematical Foundation

Since mathematics can provide clear view of an algorithm. Understanding the concepts of mathematics is aid in the design and analysis of good algorithms. Here we present some of the mathematical concepts that are helpful in our study.

Exponents

Some of the formulas that are helpful are:

$x^a \; x^b = x^{a+b}$

$x^a / x^b = x^{a-b}$

$(x^{a})^b = x^{ab}$

$x^n + x^n = 2x^n$

$2^n + 2^n = 2^{n+1}$

Logarithmes

Some of the formulas that are helpful are:

1. $\log_a b = \log_c b / \log_c a$; $c>0$
2. $\log ab = \log a + \log b$
3. $\log a/b = \log a - \log b$
4. $\log (a^b) = b \log a$
5. $\text{Log } x < x$ for all $x>0$
6. Log 1 = 0, log 2 = 1, log 1024 = 10.
7. $_a\log b^n = {}_n\log b^a$

Series

1. $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$
2. $\sum_{i=0}^{n} a^i \leq 1 / 1\text{-}a$; if $0<a<1$

   $= a^{n+1} - 1 / a\text{-}1$; else

3. $\sum_{i=1}^{n} i = n(n+1) / 2$
4. $\sum_{i=0}^{n} i^2 = n(n+1)(2n+1) / 6$
5. $\sum_{i=0}^{n} i^k \approx n^{k+1} / |k+1|$; $k \mathrel{!}= -1$
6. $\sum_{i=1}^{n} 1/i \approx \log_e^n$

Asymptotic Notations

- The factor of time is usually more important than that of space, so efficiency considerations usually focus on the amount of time elapsed when processing data.
- To evaluate an algorithm's efficiency, real-time units such as microseconds and nanoseconds should not be used. Rather, logical units that express a relationship between the size **n** of a file or an array and the amount of time **t** required to process the data should be used.
- A function expressing the relationship between **n** and **t** is usually much more complex, and calculating such a function is important only in regard to large bodies of data; any terms that do not substantially change the function's magnitude should be eliminated from the function. The resulting function gives only an approximate measure of efficiency of the original function. However, this approximation is sufficiently close to the original, especially for a function that processes large quantities of data. This measure of efficiency is called **asymptotic complexity**, and is used when disregarding certain terms of a function to express the efficiency of an algorithm or when calculating a function is difficult or impossible and only approximations can be found.
- The word **asymptotic** means approaching a value or curve arbitrarily closely. When it comes in terms of analyzing the complexity in terms of time and space, we can never provide an exact number to define the time and space required by algorithm. Instead we express it using standard notation which is called **asymptotic notation**. (Examples: **Big – Oh, Big - Omega, Big Theta**).

Note: To estimate running time of an algorithm we should concentrate about growth rate of time on increasing size of input to the algorithm but not about the absolute time required for it. So it is convenient to say estimating the bounding time and bounding space. The bounding time and bounding space for an algorithm to execute are represented by using some well-known mathematical notation called asymptotic notations.

Big-O Notation

The most commonly used notation for specifying asymptotic complexity—that is, for estimating the rate of function growth — is the big-O notation introduced in 1894 by Paul Bachmann.

Given two non-negative valued functions **f** and **g**, the function **f(n)** is said to be Big Oh of function **g(n)** and written as **f(n) = O(g(n))** or simply **f = O(g)** if there exist some positive constant **c** and $\mathbf{n_0}$ such that $\mathbf{f(n) \leq cg(n)}$ $\forall$ $\mathbf{n \geq n_0}$

— **O** Notation provides an **asymptotic upper bound** on a function.

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}.$$

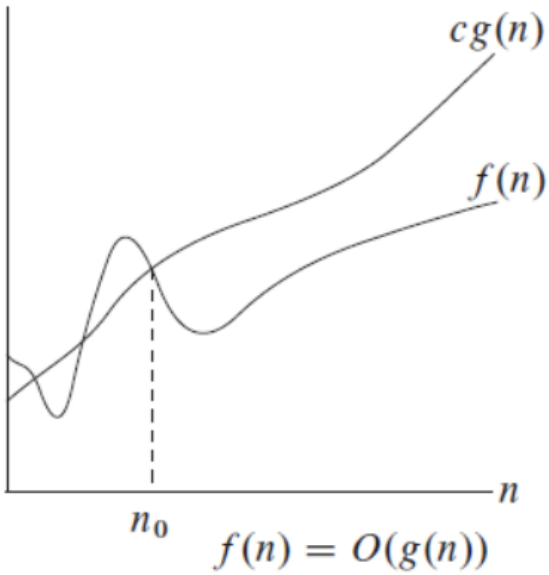


Figure 1: Graphical representation of Big Oh notation

Big O-notation **gives an upper bound** for a function to within a constant factor.

We write **f(n) = O(g(n))** if there are positive constants $\mathbf{n_0}$ and **c** such that at and to the right of $\mathbf{n_0}$, the value of **f(n)** always lies on or below **cg(n) .**

Properties of Big-Oh Notation

1. If f (n) is O(g(n)) and g(n) is O(h(n)), then f (n) is O(h(n)).
2. If f(n) is O(h(n)) and g(n) is O(h(n)), then f (n) + g(n) is O(h(n)).
3. The function an^k is $O(n^k)$.
4. The function n^k is $O(n^{k+j})$ for any positive j.
5. If f (n) = cg(n), then f (n) is O(g(n)).
6. The function $\log_a n$ is $O(\log_b n)$ for any positive numbers a and $b \neq 1$.

7. $\log_a n$ is O(lg n) for any positive $a \neq 1$, where $\lg n = \log_2 n$

Note: O(1) is used to denote constants.

Example:

$f(n) = 2n^2 + 3n + 1$ and $g(n) = n^2$

To say $f(n) = O(g(n))$, we have to show $f(n) \leq cg(n)$.

i.e. $2n^2 + 3n + 1 \leq c\, n^2$

If we choose, $n_0 = 1$ and $c = 6$ then

$2n^2 + 3n + 1 \leq 6\, n^2 \quad \forall\, n \geq 1$

Which is true, so we can say $f(n) = O(g(n))$, i.e $f(n) = O(n^2)$

Big-Ω Notation

Given two non-negative valued functions **f** and **g**, the function **f(n)** is said to be Big Omega of function **g(n)** and written as **f(n) = Ω (g(n))** or simply **f = Ω (g)** if there exist some positive constant **c** and $\mathbf{n_0}$ such that **f(n) ≥ cg(n) ∀ n ≥ n_0**

— Ω Notation provides an asymptotic **lower bound** on a function.

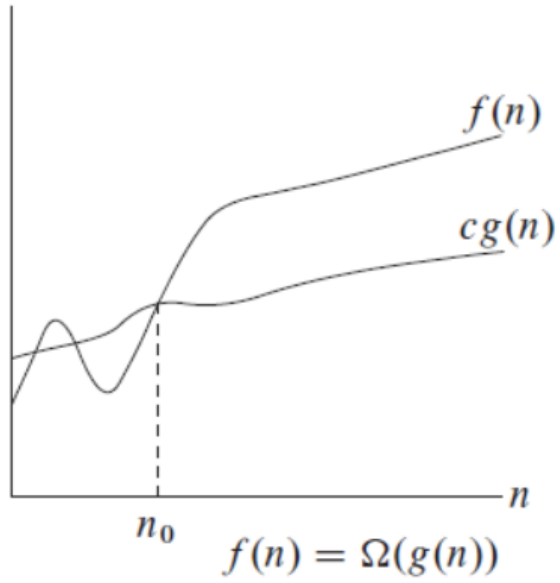


Figure 2: Graphical representation of Big Omega notation

Big Omega notation gives a **lower bound** for a function to within a constant factor. For all values n at or to the right of n_0, the value of **f(n)** is on or above **cg(n)**.

Example:

f(n) = 3n + 5 and g(n) = n

To say f(n) = Ω (g(n)), we have to show f(n) ≥ cg(n).

i.e. 3n +5 ≥ c n

If we choose, n_0 =1 and c= 2 then

3n +5 ≥ 2 n ∀ n ≥ 1

Which is true, so we can say f(n) = Ω (g(n)), i.e f(n) = Ω (n)

Big- Θ Notation

Given two non-negative valued functions **f** and **g**, the function **f(n)** is said to be Big Theta of function **g(n)** and written as **f(n) = Θ (g(n))** or simply **f = Θ (g)** if there exist some positive constant $\mathbf{c_1}$, $\mathbf{c_2}$ and $\mathbf{n_0}$ such that $\mathbf{c_1 g(n) \leq f(n) \leq c_2 g(n)}$ **∀ n ≥ $\mathbf{n_0}$**

— Θ Notation provides an asymptotically **tight bound** on a function.

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}.$$

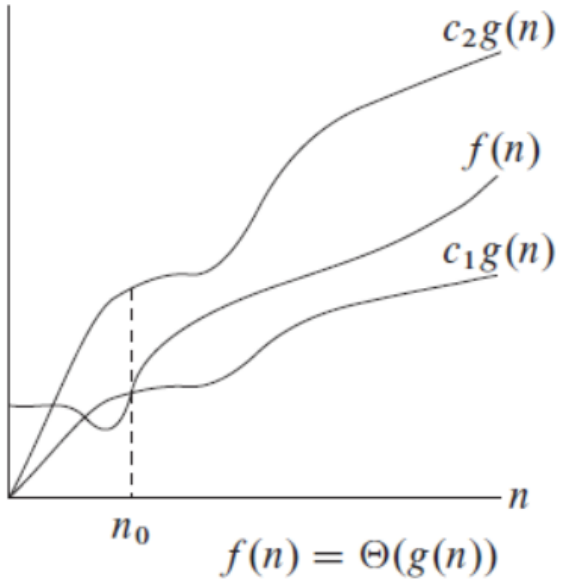


Figure 3: Graphical representation of Big Theta notation

Θ,-notation bounds a function to within constant factors. We write **f(n) = Θ (g(n))** if there exist positive constants **c_1** , **c_2** and **n_0** such that at and to the right of n_0, the value of **f(n)** always lies between **c_1g(n)** and **c_2g(n)** inclusive.

If f(n) = O(g(n) and f(n) = Ω(g(n)) then f(n) = Θ(g(n))

Example:

f(n) = $3n^2 + 4n + 7$ and g(n) = n^2

To say f(n) = Θ (g(n)), we have to show f(n) ≥ c_1g(n) and f(n) ≤ c_2g(n)

i.e. $3n^2 + 4n + 7 \ge c_1n^2$ and $3n^2 + 4n + 7 \le c_2n^2$

If we choose n_0 =1 and c_1= 3 then

$3n^2 + 4n + 7 \geq 3n^2$

Which is true, so we can say $f(n) = \Omega (g(n))$

Similarly,

If we choose, $n_0 = 1$ and $c_2 = 14$ then $3n^2 + 4n + 7 \leq 14\, n^2 \quad \forall\, n \geq 1$

Which is true, so we can say f(n) = O(g(n))

Here,

f(n) = O(g(n) and f(n) = Ω(g(n))

So, f(n) = Θ(g(n)) i.e. f(n) = Θ(n^2)

Examples of Complexities

Algorithms can be classified by their time or space complexities, and in this respect, several classes of such algorithms can be distinguished.

For example, an algorithm is called **constant** if its **execution time remains the same for any number of elements**; it is called **quadratic** if **its execution time is O(n^2).**

Some complexity classes with their notations are given below. [n is input size]

- **Constant** ---------- O(1)
- **Logarithmic** ---------- O(lg n)
- **Linear** ---------- O(n)
- **Logarithmic Linear** ------ O(n lg n)
- **Quadratic** ---------- O(n^2)
- **Cubic** ----------O(n^3)
- **Exponential** ---------- O(2^n)

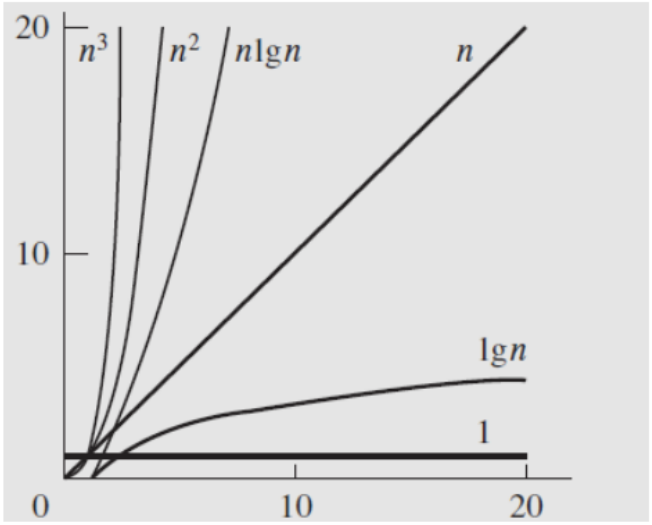


Figure 4: Growth of different functions (x axis is input size and y axis is time

The Best, Average, and Worst Cases

Best case complexity

Best case complexity gives the lower bound on the running time of an algorithm for any instance of input n. This indicates that, the algorithm can never have lower running time than the best case complexity for a particular class of problems.

Example:

In linear search, searching key element present at first index. Best case time complexity O(1).

Worst case complexity

Worst case complexity gives the upper bound on the running time of an algorithm for all instance of input n. It provides guarantee that the algorithm will never take any longer time than worst case complexity. Worst case complexity gives the longest running time for any input of size n.

Example:

In linear search, searching key element present at last index. Worst case time complexity O(n).

Average case complexity

Average case complexity gives the average number of steps required for any instance of input for the algorithm. The average case falls between the best and worst case. In simple cases, the average complexity is established by considering possible inputs to an algorithm, determining the number

of steps performed by the algorithm for each input, adding the number of steps for all the inputs, and dividing by the number of inputs.

Detailed Analysis of algorithm

The process of finding complexity of given algorithm by counting the number f steps and number of memory references used by using RAM model is called detailed analysis For detailed time complexity of algorithm we count the number of steps and finally add all steps and calculate their big oh notation. Similarly for detailed analysis of space complexity we count the number of memory references and finally add all steps and find their big oh notation.

Example: Compute the time and space complexity of following algorithm

```
int sum(int   n){
        int total =0;
        for(int i=1;i<=n;i++){ total =
                total+i*i;
        }
        return total;
}
```

Time Complexity

total =0; takes 1 step

In for loop,

i=1 takes 1 step

i<=n takes n+1 step

i++ takes n step

total = total+i*i takes 3n steps [1 multiplication, 1 addition and

1 assignment occurs n time] return

statement takes 1 step

Total number of steps = 1 + 1 + (n+1) + n + 3n + 1

= 5n+ 4

So, Time Complexity = O(n)

Space Complexity

Total number of memory reference = 3 (space for variables n, i and total)

So, Space complexity = O(1)

Note: Only the dominant term is considered while writing asymptotic notation (constant and other terms are ignored)

Example 2: Compute the time complexity of following algorithm.

```
for(i = 0; i<n; i++){
      for (j = 0; j<n; j++){
      sum = i*j +sum;
      }
}
```

In outer loop,

i = 0 step count = 1

i<n step count = n+1

i++ step count = n

In inner loop,

j= 0, step count = 1

j<n , step count = n+1

j++, step count = n

sum = i*j +sum , step count = 3n

All steps inside inner loop occurs n time so,

There are n((1)+ (n+1) + (n) + (3n)) = n(5n +2) steps.

Here, total step count = (1) + (n+1) + (n) + (n(5n+2))

$= 5n^2 + 4n + 2$

Hence, the asymptotic complexity in Big Oh notation is **$O(n^2)$**.

Space Complexity

Total number of memory reference = 4 (space for variables i,j,n, and sum)

So, Space complexity = O(1)

Example: Detailed analysis of algorithm to find factorial of given integer.

```
int factorial(int  n){
       int fact =1;
      for (int i =1; i<=n;i++){
             fact = fact*i;
       }
        return fact;
}
```

Time Complexity

fact = 1 takes 1 step.

in for loop,

i=1 takes 1 step

i<=n takes (n+1) steps

i++ takes n steps

within for loop fact = fact*i takes 2n steps (one multiplication and one assignment n time)

return statement takes 1 step

total steps = 1 + 1+ (n+1) + n +2n +1

= 4n+4

So, time complexity is = O(n)

Space Complexity

Total number of memory references = 3 (three variables: fact, i, n)

So, space complexity = O(1)

Example: Bubble Sort algorithm analysis

```
BubbleSort(a,  n){
      for (i = 1; i < n; i++) {
            for (j = 0; j < n - i ; j++) {
                  if (a[j] > a[j + 1]) { temp = a[j];
                        a[j] = a[j + 1]; a[j + 1] =
                        temp;
                  }
            }
      }
}
```

Time complexity

In outer for loop,

i =1 takes 1 step

i<n takes n steps

i++ take n-1 steps

In inner for loop,

j=0 takes n steps

j< n-i takes [n+ (n-1)+(n-2) + . . . +3+2+1] , say n+k steps

j++ takes [(n-1) + (n-2)+(n-3) + . . . +3+2+1], say k steps

In if statement

a[j] > a[j + 1]) takes at most k steps

temp = a[j], a[j] = a[j + 1] and a[j + 1] = temp takes 1/1 steps altogether 3. So if statement takes 3k steps

So total step counts = 1 + (n) + (n-1) +(n+k) + k + 3k

= 3n + 5k

= 3n + 5[(n-1) + (n-2)+(n-3) + . . . +3+2+1]

= 3n + 5[1+2+3+ . . . + (n-3) + (n-2)+(n-1)+n] -5n

$= -2n + 5[\frac{n}{2}(n+1)]$ (Sum of first n natural numbers)

$= \frac{5}{2}n^2 + \frac{5}{2}n - 2n$

$= \frac{5}{2}n^2 + \frac{1}{2}n$

So, Time complexity is $O(n^2)$

Space Complexity

Total number of memory references = 1 + 1 + 1+1+n (for a, n references and for i,j,n,temp 1/1 reference)

= 4+n

So, Space complexity = O(n).

Amortized Complexity

In many situations, data structures are subject to a sequence of operations rather than one operation. In this sequence, one operation possibly performs certain modifications that have an impact on the run time of the next operation in the sequence. One way of assessing the worst case run time of the entire sequence is to add worst case efficiencies for each operation. But this may result in an excessively large and unrealistic bound on the actual run time. To be more realistic, *amortized analysis can be used to find the average complexity of a worst case sequence of operations.*

- Amortized Analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyze a sequence of operations and guarantee a worst case average time which is lower than the worst case time of a particular expensive operation.
- Amortized complexity analysis is most commonly used with data structures, which have state that persists between operations. The basic idea is that an expensive operation can alter the state so that the worst case cannot occur again for a long time, thus amortizing its cost.

There are generally following three methods to compute the amortized cost (amortized analysis)

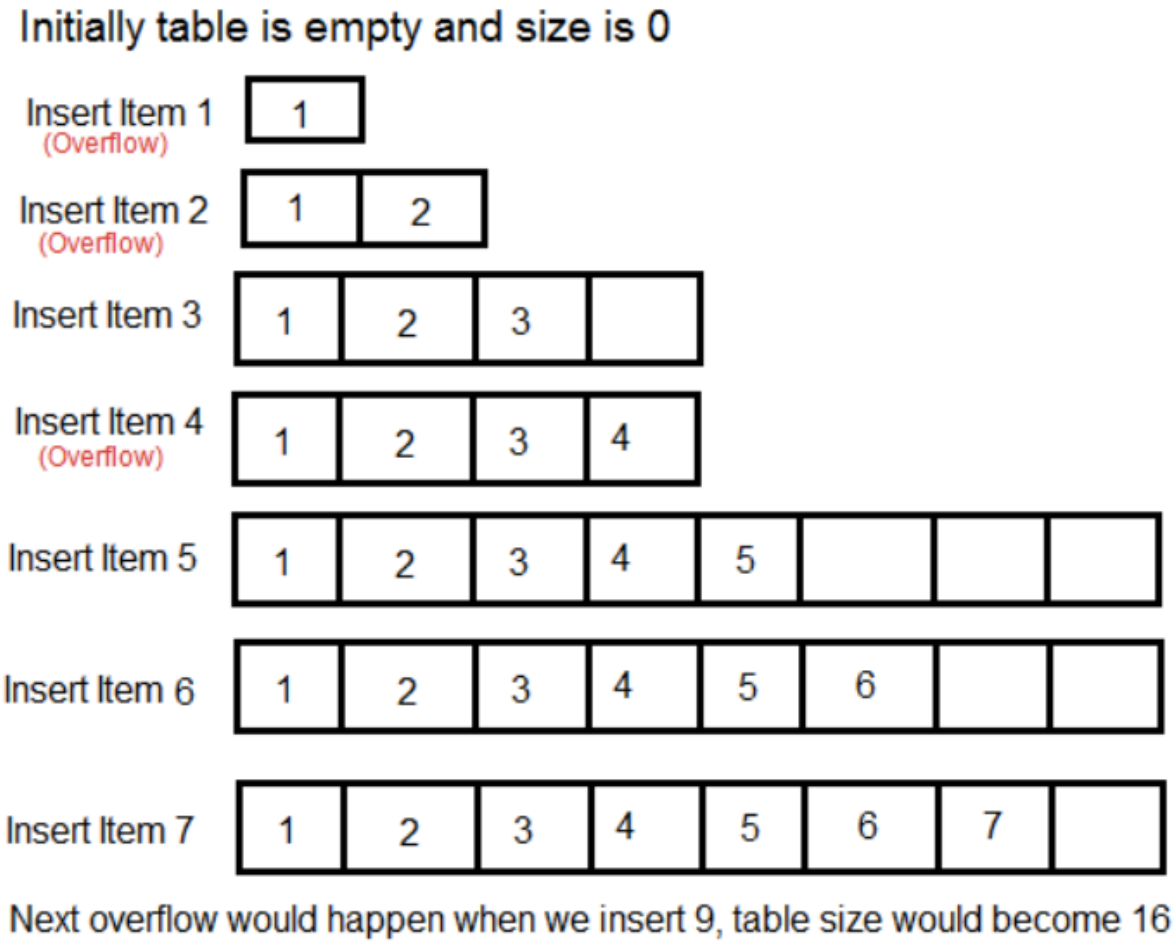
1. Aggregate Method
2. Accounting Method
3. Potential Method

The aggregate method (Aggregate Analysis)

- In the aggregate method of amortized analysis, we show that for all n, a sequence of n operations takes worst-case time T(n) in total. In the worst case, the average cost, or amortized cost, per operation is therefore T(n) / n.
- Note that this amortized cost applies to each operation, even when there are several types of operations in the sequence. The other two methods we shall study in this chapter, the accounting method and the potential method, may assign different amortized costs to different types of operations.

Example: Let's take dynamic array (table) as an example. If the array has space available, we simply insert new item in available space. If not following steps are performed:

- Allocate memory for a larger array of size twice as the old one
- Copy the contents of old array to new one and
- Free the memory of old array.



Consider a sequence of n insertions. If we use simple analysis the worst-case time to execute one insertion is $\Theta(n)$. So the worst-case time for n insertions is $n.\ \Theta(n). = \Theta(n^2)$. This analysis gives an upper bound, but not a tight upper bound for n insertions as all insertions don't take $\Theta(n)$ time.

In amortized Analysis,

The cost of i-th insertion is c_i

$$c_i = 1 + \begin{cases} i-1 & \text{if } i-1 \text{ is a power of 2} \\ 0 & \text{otherwise} \end{cases}$$

Then, the cost of n insertions is:

$$\begin{aligned} &= \sum_{i=1}^{n} c_i \\ &\leq n + \sum_{j=0}^{\lfloor lgn \rfloor} 2^j \\ &\leq 3n \\ &\in \theta(n). \end{aligned}$$

Hence the average cost of insertion of an item in dynamic array (table) is $\Theta(n)/n = \Theta(1)$.

The accounting method

In the accounting method of amortized analysis, we assign differing charges to different operations, with some operations charged more or less than they actually cost. The amount we charge an operation is called its amortized cost. When an operation's amortized cost exceeds its actual cost, the difference is assigned to specific objects in the data structure as credit. Credit can be used later on to help pay for operations whose amortized cost is less than their actual cost. Thus, one can view the amortized cost of an operation as being split between its actual cost and credit that is either deposited or used up. This is very different from the aggregate method, in which all operations have the same amortized cost.

The potential method

Instead of representing prepaid work as credit stored with specific objects in the data structure, the potential method of amortized analysis represents the prepaid work as "potential energy,"or

just "potential," that can be released to pay for future operations. The potential is associated with the data structure as a whole rather than with specific objects within the data structure.

Recursive Algorithms

- *Sometimes it is difficult to define an object explicitly. However, it may be easy to define this object in terms of itself. This process is called recursion. We can use recursion to define sequences, functions, and sets*
- **Recursion** is a method where the solution to a problem depends on solutions to smaller instances of the same problem
- *An algorithm is called recursive if it solves a problem by reducing it to an instance of the same problem with smaller input.*
- **Recursive algorithm** is a method of simplification that divides the problem into sub-problems of the same nature. The result of one recursion is the input for the next recursion. The repletion is in the self-similar fashion.
- A **recursive algorithm** is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input. More generally if a problem can be solved utilizing solutions to smaller versions of the same problem, and the smaller versions reduce to easily solvable cases, then one can use a recursive algorithm to solve that problem. For example, the elements of a recursively defined set, or the value of a recursively defined function can be obtained by a recursive algorithm
- Recursive algorithms have two cases: a recursive case and base case.
- Any function that calls itself is recursive function.
- Examples of recursive algorithms: Generation of factorial, Fibonacci number series,etc.
- In general, recursive computer programs require more memory and computation compared with iterative algorithms, but they are simpler and for many cases a natural way of thinking about the problem.

Recurrence Relations

Recurrence relation is an equation or inequality that defines a function in terms of its values on smaller inputs. A recurrence can be used to represent the running duration of an algorithm that comprises a recursive call to itself.

- A recurrence relation, T(n), is a recursive function of an integer variable n.
- Example:

$$T(n) = \begin{cases} a & \text{if } n = 1 \\ 2T(n/2) + bn + c & \text{if } n > 1 \end{cases}$$

- The portion of the definition that does not contain T is called the **base case** of the recurrence relation; the portion that contains T is called the **recurrent or recursive case**.
- Recurrence relations are useful for expressing the running times (i.e., the number of basic operations executed) of recursive algorithms
- The specific values of the constants such as a, b, and c (in the above recurrence) are important in determining the exact solution to the recurrence. Often however we are only concerned with finding an asymptotic upper bound on the solution. We call such a bound an *asymptotic solution* to the recurrence.
- Solving a recurrence relation means to obtain a function defined on the natural number that satisfy the recurrence.
- To solve recursive algorithms we need to define their recurrence relation and by using any one of the recurrence relation solving method, we calculate their complexity.
- *Cost of solving recursive algorithm = cost of dividing problem+ cost of solving sub problems +cost of merging solutions*

The recurrence relation are used to model the time or space complexity of an algorithm that uses that uses divide and conquer or recursive approach to solve that problem.

For example, if a problem is divided into 1/3 and 2/3 size of its original size on the division process by the algorithm and if the size of problem is n and it takes linear time to merge the solution of its components the time complexity of algorithm can be expressed by recurrence

relation as,

$$T(n) = T(2n/3)+T(n/3)+ O(n)$$

Example:

The Binary search algorithm divides the input array of size n into half i.e. n/2 and the half part of the array is ignored in each step and the rest half is proceeded. So the time complexity of binary search algorithm can be expressed by following recurrence relation

$T(n) = T(n/2) + O(1)$ for $n>1$

$= 1$ for $n = 1$

Examples:

For finding factorial

$T(n)=1$	when $n =1$
$T(n)=T(n-1) + O(1)$	when $n>1$

For finding Nth Fibonacci number

$T(1)=1$	when $n=1$
$T(2)=1$	when $n=2$
$T(n)=T(n-1) + T(n-2) +O(1)$	when $n>2$

Solving Recurrences

- The process of finding asymptotic bounds on the solution.
- Solving recurrence means finding the solution of given recurrence relation in terms of Big Oh Notation.
- Some of the techniques to solve recurrences
    - Recursion tree method
    - Iterative expansion method
    - Substitution method
    - Master method

Substitution Method

- In the substitution method, we guess a bound and then use mathematical induction to prove our guess correct.
- The substitution method for solving recurrences comprises two steps:
    1. Guess the form of the solution.
    2. Use mathematical induction to find the constants and show that the solution works.

  We substitute the guessed solution for the function when applying the inductive hypothesis to smaller values; hence the name "substitution method." This method is powerful, but we must be able to guess the form of the answer in order to apply it.

Unfortunately, there is no general way to guess the correct solutions to recurrences. Guessing a solution takes experience and, occasionally, creativity. Fortunately, though, you can use some heuristics to help you become a good guesser. You can also use recursion trees. If a recurrence is similar to one you have seen before, then guessing a similar solution is reasonable.

Example: Solve the following recurrence relation by using substitution method.

$T(n) = 1$ $\quad n=1$

$T(n) = 4T(n/2) + n$ $\quad n>1$

Let's Guess $T(n) = O(n^3)$

$\Rightarrow \quad T(n) \le cn^3 \quad \forall n > n_0$ ---------- (1)

Now prove this by mathematical induction as

Base step, For n=1:

$T(n) = c*1^3$ $\quad$ Definition

$1 \le c$ $\quad$ which is true for all positive values of c

Inductive step, Let's assume it is true for $\forall_k < n$

Then $T(k) \le ck^3$ ----------(2)

It is also true for k = n/2

Now eq^n (2) becomes

$T(1/2) \le c(n/2)^3$

$\le c(n^3/8)$

Now,

$T(n) = 4T(n/2) + n$

$\leq 4c(n^3/8) + n$

$= c(n^3/2) + n$

$= cn^3 - n(cn^2/2 - 1) \leq cn^3$

Hence eq^n (1) is true.

So, $T(n) = O(n^3)$

Iterative expansion Method

- Also called as iterative method, backward substitution iterative substitution

In this method, the given recurrence relation is expanded mathematically and the summation of expanded sequence of terms is obtained in order to get estimated solution of the function.

— Expand the relation so that summation independent on n is obtained.

— Bound the summation

This method requires mathematical knowledge for getting the solution.

Example: Solve the following recurrence relation by using iterative expansion method

$T(n)= 2T(n/2) +1$ when $n>1$

$T(n)= 1$ when $n=1$

Solution,

$T(n) = 2T(n/2) +1$

$= 2 \{ 2T(n/4) + 1\} +1$ [Here $T(n/2) = 2T(n/4) + 1$]

$= 4T(n/4) + 2 + 1 = 2^2 T(n/2^2) +2^1 +1$

$= 4 \{ 2T(n/8) +1 \} + 2 + 1$ [Here $T(n/4) = 2T(n/8) + 1$]

$= 8 T(n/8) + 4 + 2 + 1 = 2^3T(n/2^3) +2^2 + 2^1 + 1$

Expanding similarly upto kth term

$T(n) = 2^kT(n/2^k) +2^{k-1}+ 2^{k-2} + +.....+ 2^2 + 2^1 + 2^0$

For simplicity assume:

$n= 2^k$

Then , $k=\log n$

So, $T(n) = n T(1) + 2^{k-1}+ 2^{k-2} + + ...+ 2^2 + 2^1 + 2^0$

$= n T(1) + (2^0 + 2^1 + 2^2 +..... 2^{k-2}+ 2^{k-1})$

$= n + (2^k - 1)$ [after computing sum of above geometric series]

$= n + n - 1$

$= 2n - 1$

So, $T(n) = O(n)$.

Exercise: Solve the following recurrence relation by using iterative expansion method

$T(n) = 2T(n/2) + kn$ when $n>1$

$T(n) = 1$ when $n=1$ where k is a constant

(Answer: O(nlogn)

Recursion Tree Method

- In recurrence tree method, the given recurrence relation is expanded into a tree and summation of nodes at each level is obtained.
- The recursion-tree method converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence
- In a recursion tree, each node represents the cost of a single sub problem somewhere in the set of recursive function invocations. We sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level costs to determine the total cost of all levels of the recursion
- In general we consider non recursive term of recurrence as the root of recurrence tree.

Process:

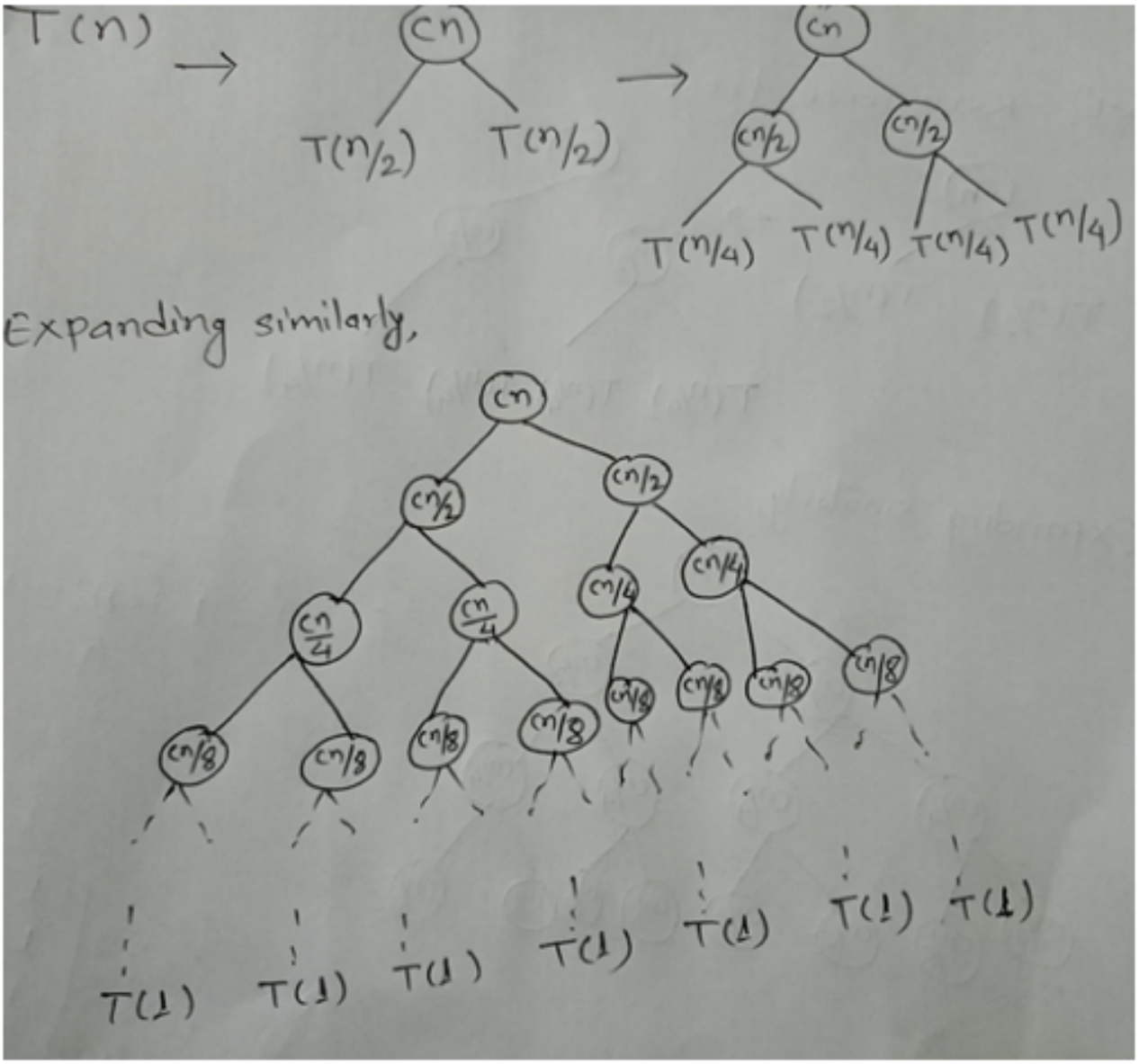
- *Draw a recursion tree based on the given recurrence relation.*
- *Determine:*
  - *Cost of each level*
  - *Total number of levels in the recursion tree*
  - *Number of nodes in the last level*
  - *Cost of the last level*
- *Add cost of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation*

Example: Solve following recurrence relation using recurrence tree method.

$T(n) = 1$ $\quad n=1$

$T(n) = 2T(n/2) + cn$ $\quad n>1$

Drawing a recursion tree based on the given recurrence relation.



Determining the cost of each level-

Cost of level-0 = cn

Cost of level-1 = cn/2 + cn/2 = cn

Cost of level-2 = cn/4 + cn/4 + cn/4 + cn/4 = cn and so on.

Determining the total number of levels in the recursion tree

Size of sub-problem at level-0 = $n/2^0$

Size of sub-problem at level-1 = $n/2^1$

Size of sub-problem at level-2 = $n/2^2$

Continuing in similar manner, we have-

Size of sub-problem at level-i = $n/2^i$

Suppose at level-k (last level), size of sub-problem becomes 1. Then-

$n / 2^k = 1$

$\Rightarrow \quad 2^k = n$

Taking log on both sides, we get-

$k\log_2 n = \log_2 n$

$\therefore \ k = \log_2 n$

$\therefore$ Total number of levels in the recursion tree = $k+1 = \log_2 n + 1$

Determining number of nodes in the last level

Level-0 has 2^0 nodes i.e. 1 node

Level-1 has 2^1 nodes i.e. 2 nodes

Level-2 has 2^2 nodes i.e. 4 nodes

Continuing in similar manner, we have-

Level-k has 2^k nodes i.e. $2^{\log_2 n} = n$ nodes

Determining the cost of last level-

Cost of last level = $n \times T(1) = n \times 1 = O(n)$

Total cost

$T(n) = (cn+cn+cn+cn+ \ldots\ldots)$ [k times i.e. $\log_2 n$ times] + $O(n)$

$= cn \log_2 n + O(n)$

$= O(n \log_2 n)$

Master Method

The master method provides a "cookbook" method for solving recurrences of the form

T(n) = a T(n/b) + f(n)

Where a≥1, b>1, f(n) asymptotically positive function.

This method is based on Masters Theorem .

Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Idea:

In each case, we compare the function f(n) with $\boldsymbol{n^{\log_b a}}$ and the solution of recurrence is derived from the larger of the two functions.

Case 1: When $n^{\log_b a} > f(n)$

$T(n) = \Theta(n^{\log_b a})$

Case 2: When $n^{\log_b a} = f(n)$

$T(n) = \Theta(n^{\log_b a} * \log_2 n)$

Case 3: When $n^{\log_b a} < f(n)$

$T(n) = \Theta(f(n))$

Example:

$$T(n) = 2T(n/2) + n \quad , n>1$$
$$= 1 \quad , n=1$$

Comparing $T(n) = 2T(n/2) + n$ with $T(n) = a\,T(n/b) + f(n)$

$a = 2, b = 2, f(n) = n$

$n^{\log_b a} = n^{\log_2 2} = n$

Here, $n^{\log_b a} = f(n)$ (Case2)

$\therefore T(n) = \Theta(n \lg n)$

Example:

$$T(n) = 9T(n/3) + n \quad , n>1$$
$$= 1 \quad , n=1$$

Comparing $T(n) = 9T(n/3) + n$ with $T(n) = a\,T(n/b) + f(n)$

$a = 9, b = 3, f(n) = n$

$n^{\log_b a} = n^{\log_3 9} = n^2$

Here, $n^{\log_b a} > f(n)$ (Case1)

$\therefore T(n) = \Theta(n^2)$

Exercise:

Use the master method to give tight asymptotic bounds for the following recurrences.

a. $T(n) = 2T(n/4) + 1.$

b. $T(n) = 2T(n/4) + \sqrt{n}.$

c. $T(n) = 2T(n/4) + n.$

d. $T(n) = 2T(n/4) + n^2.$