Revision - 1st Internals MODULE 1

Class	DAA
Created	@Feb 27, 2021 10:14 PM
Materials	https://drive.google.com/file/d/11T8Rvqy8L86lcxiMdYKDbx0I0TmzYI6y/view

Introduction

Algorithm

An algorithm is a finite set of instructions that accomplishes a particular task. Another definition is a sequence of unambiguous instructions for solving a problem in a finite amount of time.

All algorithms must satisfy the following conditions:

1. Input: Zero or more quantities are externally supplied as input.
2. Output: At least one quantity is produced by given algorithm as output.
3. Definiteness: Each instruction is clear and unambiguous
4. Effectiveness: Each instruction must be very basic, so it can be carried out by a person using a pen and paper.
5. Finiteness: If we can trace the instructions of the algorithm then for all cases, the algorithm must terminate after a finite number of steps.

Process for Design and Analysis of Algorithms

1. **Understand the problem.** This is a very crucial phase, and the correctness of the algorithm relies on proper understanding of the problem.
2. **Solution as an algorithm.** Solve the problem exactly if possible, or with approximation method if not.
3. **Algorithm techniques.** In this we use different design techniques like
    - Divide and Conquer
    - Greedy Method
    - Dynamic Programming
    - Backtracking
    - Branch and bound etc

4. **Prove correctness.**
5. **Analyze the algorithm.** This means studying the algorithm behavior, calculating time complexity, space complexity etc. If the time complexity is too high or it is otherwise inefficient, we use one of the algorithm techniques to refine it to make it more efficient.
6. **Code the algorithm.** After solving all the prior phases successfully we then code the algorithm. The code should not rely on any one programming language, so we use pseudo-code for the same.

Algorithm Design Paradigms

There are essentially 5 fundamental techniques used to design an algorithm efficiently:

1. Divide and Conquer
2. Greedy method
3. Dynamic Programming
4. Backtracking
5. Branch and bound.

Divide and Conquer.

It is a top-down approach to solve a problem. The algorithm of divide and conquer technique follows 3 steps:

- Divide the original problem into a set of sub-problems
- Conquer (or solve) each sub-problem individually, recursive.
- Combine the solution of these sub-problems to get the solution of the original problem.

Greedy Technique

This technique is used to solve an optimization problem. An optimization problem is one in which we are given a set of input values which are to be either maximized or minimized wrt some constraints or conditions.

Greedy algorithm always makes the choice that looks the best at the moment to optimize a given function. It makes a locally optimal choice in the hope that it will lead to overall globally optimal solution.

It does not always guarantee the most optimal solution but it generally produces solutions that are close to the most optimal.

Dynamic Programming

It is similar to divide and conquer technique in that they both break down the original problem into smaller sub-problems that can be solved recursively. The difference is that in dynamic programming, the results obtained from solving the smaller sub-problems are reused by maintaining a table of results in the calculation of larger sub-problems.

Thus it is a bottom-up approach that begins by solving the smaller sub-problems, saving these partial results, and then using the aforementioned to solve the larger sub-problems. This is repeated until the solution to the original problem is solved.

It takes much lesser time than naïve or straightforward methods such as divide-and-conquer.

Backtracking

This can only be applied on problems that admit the concept of a partial candidate solution and a relatively quick test of whether it can be completed to a complete solution.

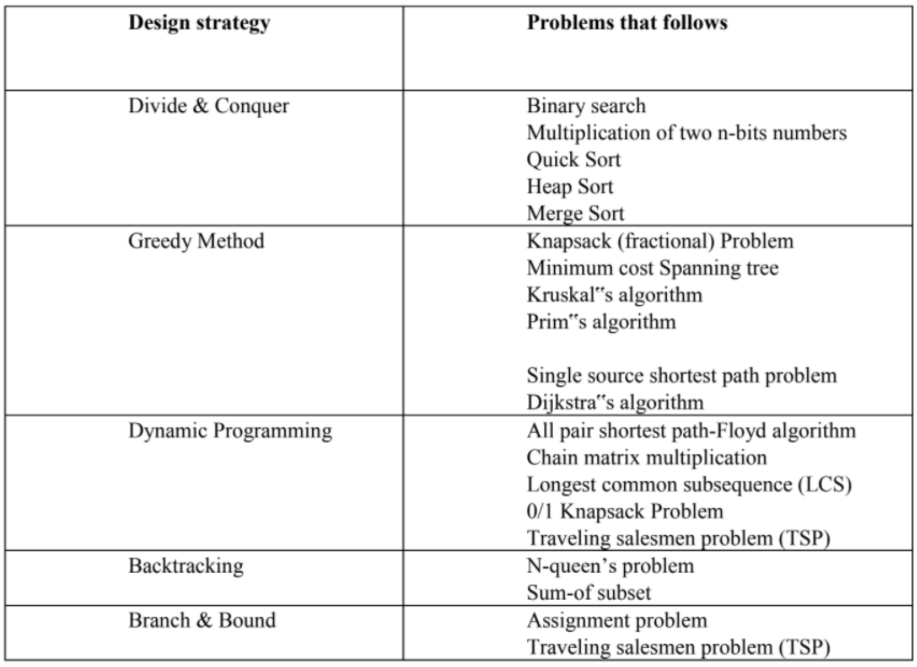
These algorithms try each possible solution until it finds the right one, it is a depth-first search of the set of possible solutions.

During the search, if one alternative doesn't work, it backtracks to the choice point and tries the next alternative. This process repeats until the right solution is found.

Breach and Bound

It is a rather general optimization technique that applies where the greedy method and dynamic programming fail. However, it is much slower.

The general idea of B&B is a BFS-like search for the optimal solution, but not all nodes get expanded. Rather a carefully selected criterion determines whether to expand a node and when, and another tells the algorithm when the optimal solution has been found.



Design strategy	**Problems that follows**
Divide & Conquer	Binary search Multiplication of two n-bits numbers Quick Sort Heap Sort Merge Sort
Greedy Method	Knapsack (fractional) Problem Minimum cost Spanning tree Kruskal‟s algorithm Prim‟s algorithm Single source shortest path problem Dijkstra‟s algorithm
Dynamic Programming	All pair shortest path-Floyd algorithm Chain matrix multiplication Longest common subsequence (LCS) 0/1 Knapsack Problem Traveling salesmen problem (TSP)
Backtracking	N-queen's problem Sum-of subset
Branch & Bound	Assignment problem Traveling salesmen problem (TSP)

Types of Algorithms

There are 4 types of algorithms

1. Approximate algorithms
2. Probabilistic algorithms
3. Infinite algorithm
4. Heuristic algorithm

Approximate algorithm

An algorithm is said to be approximate if it is infinite and repeating

Ex: $\sqrt{2} = 1.414$
$\sqrt{3} = 1.713$
$\pi = 3.14$ Etc...

Probabilistic algorithm

An algorithm is said to be probabilistic if the solution of a problem is uncertain.

ex: Tossing of a coin

Infinite algorithm

An algorithm which is not finite is called an infinite algorithm.

ex: Division by zero

Heuristic algorithm

Giving fewer inputs to get more outputs is known as a heuristic algorithm.

ex: All business applications.

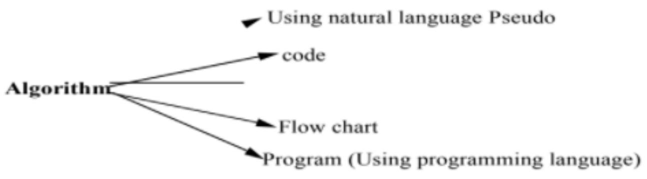
Criteria or issues for algorithms:

1. How to devise an algorithm: The creation of algorithms is a logical activity that may never be fully automated.
2. How to express algorithms: Using the best principles of structuring
3. How to validate algorithms: The process of checking an algorithm computes the correct answer for all possible legal inputs is called algorithm validation. It's purpose is to find whether an algorithm works properly without being dependent on specific programming languages.
4. How to analyze algorithms: The task of determining how much computing time and storage is required by an algorithm is known as analyzing the algorithm. The behavior of algorithm in best case, average case and worst case needs to be determined.
5. How to test a program: Consists of two phases.
   - Debugging: Ensuring that the program does not produce faulty results for correct inputs, or errors of any other sort.

- Profiling or performance measuring: This is the process of calculating time and space required by a correct program for a valid set of inputs.

Specification of Algorithm.

There are various ways in which we can specify an algorithm


Algorithm
Using natural language Pseudo
code
Flow chart
Program (Using programming language)

Pseudo-code for expressing algorithms

Pseudocode is a representation of algorithm in which instruction sequence can be given with the help of programming constructs. It however does not follow any particular programming language.

General procedure/guidelines:

1. Comments begin with // and continue to the end of the line
2. A block of statements is represented within {}
3. The delimiters (;) are used at the end of each statement
4. All identifiers begin with a letter. eg: num, num1, person1, n_times etc.
5. Assignment of values to variables is done using assignment operator like =
6. There are 2 boolean values TRUE and FALSE.

   Logical operators AND, OR, NOT

   Relational operators <,>,≤,≥,=,≠

   Arithmetic operators +,-,/,*,%
7. If conditional statements exist then it is written in the form

   If(condition) then(statement)

   if(condition) then(statement-1) else(statement-2)
8. Case statements

   case

   {

   ____

   ____

   ____

   }

9. Loops
    - for loop

      ```
      for(condition) do{
          ______
          }
      ```
    - while loop

      ```
      while(condition) do{
          ______
          }
      ```
10. Break statement is used to exit from loop
11. Elements of array are accessed using []
12. Functions can be defined and used

etc.

Performance Analysis

Performance analysis of an algorithm refers to the process of determining the efficiency of an algorithm. Two things are taken into consideration:

- Time complexity
- Space complexity

Space Complexity

The space complexity of a program is the amount of memory it requires to run until completion. The space needed by an algorithm has the following components.

Instruction Space

This is the space required to store the compiled version of program instructions. This depends upon factors such as:

- The compiler used to compile the program into instruction code
- The compiler options in effect at the time of compilation
- The target computer.

Data Space

This is the space required to store all the constant and variable values. It has two components

- Space needed by constants

- Space needed by dynamically allocated objects such as structures, classes, arrays etc.

Environmental Stack Space

This is used during execution of functions. Each time functions are involved, the following data are saved in the ESS:

- The return address
- The value of local variables
- The value of formal parameters in the function being invoked.

ESS is mainly used in recursive functions

Thus the space complexity of a program p may be written as:

Space complexity S(P) = C + Sp (Instance Characteristics)

This equation shows that the total space required by programs is divided into two parts.

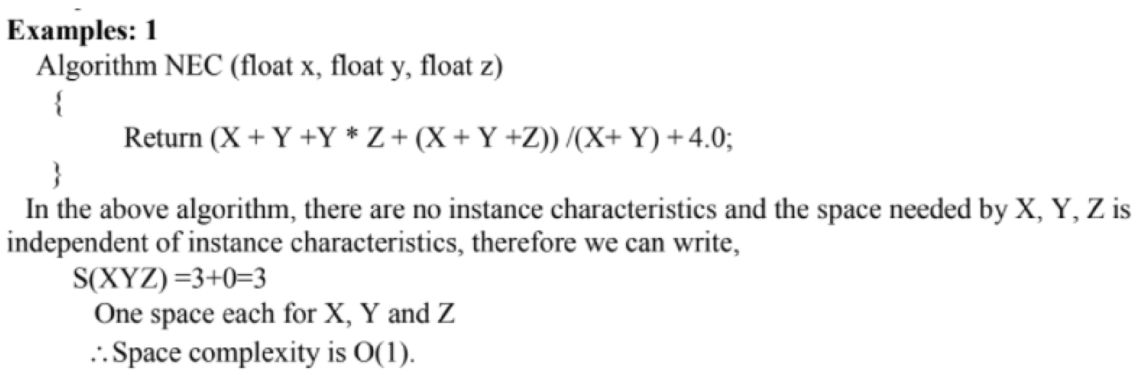
- Fixed space requirements (C) is independent of instance characteristics of the inputs and outputs

  → Instruction space

  → Space for simple variables, fixed-size structure variables, constants.

- Variable space requirements (Sp)

  This part includes dynamically allocated space and the Recursion stack space.



Examples: 1

```
Algorithm NEC (float x, float y, float z)
{
        Return (X + Y +Y * Z + (X + Y +Z)) /(X+ Y) + 4.0;
}
```

In the above algorithm, there are no instance characteristics and the space needed by X, Y, Z is independent of instance characteristics, therefore we can write,

S(XYZ) =3+0=3

One space each for X, Y and Z

∴ Space complexity is O(1).

Time Complexity

The time complexity of an algorithm is the amount of compile time required for it to run till completion. We measure the time complexity in two approaches:

- Priori analysis or *compile time*
- Posteriori analysis or *run time*

In priori analysis , before the program is executed we analyze the behavior of the algorithm. It concentrates on determining the order of execution of statements.

In posteriori analysis, we measure the execution time while the algorithm is running. It gives accurate values but is very costly.

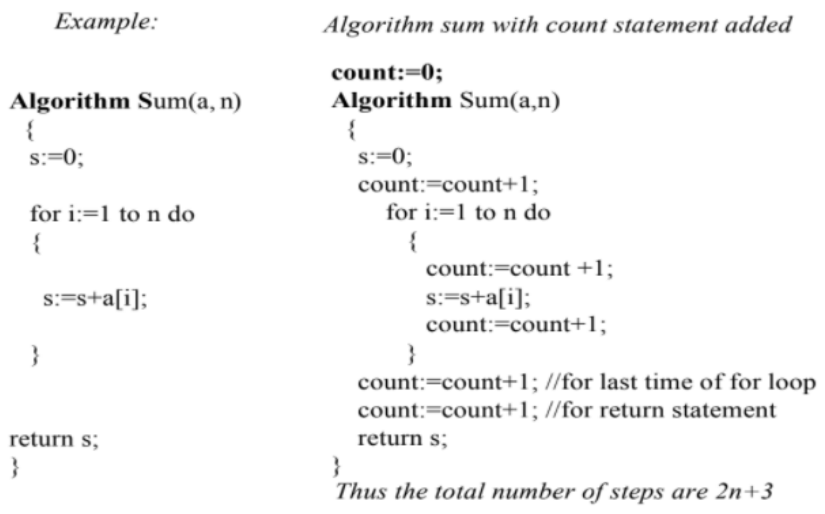
We know that compile time does not depend on the size of the inputs, hence we limit ourselves to only consider the run-time which depends on the size of the inputs, denoted as TP(n)

Time complexity T(P) = C +TP(n)

Another method is by using step count. By using stepcount, we can determine the number of steps needed by a program to solve a particular problem. We can do this in 2 ways

Method 1

We introduce a global variable named "count" and initialized to 0. So each time an instruction in the signal problem is executed, the count value is incremented by the step count of that statement.



Example:

```
Algorithm Sum(a, n)
 {
  s:=0;

  for i:=1 to n do
  {

   s:=s+a[i];

  }


return s;
}
```

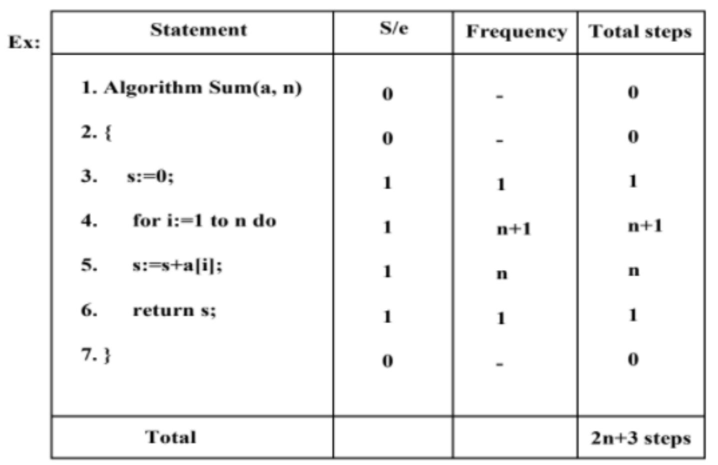
Algorithm sum with count statement added

```
count:=0;
Algorithm Sum(a,n)
 {
   s:=0;
   count:=count+1;
     for i:=1 to n do
       {
         count:=count +1;
         s:=s+a[i];
         count:=count+1;
       }
   count:=count+1; //for last time of for loop
   count:=count+1; //for return statement
   return s;
}
```

Thus the total number of steps are 2n+3

Method 2

The second method is to build a table in which we list the number of steps contributed by each statement.



Ex:

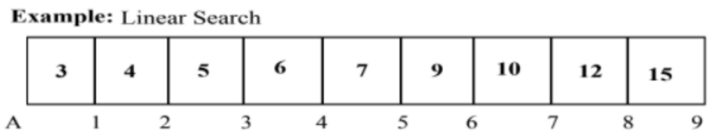
Statement	S/e	Frequency	Total steps
1. Algorithm Sum(a, n)	0	-	0
2. {	0	-	0
3. s:=0;	1	1	1
4. for i:=1 to n do	1	n+1	n+1
5. s:=s+a[i];	1	n	n
6. return s;	1	1	1
7. }	0	-	0
Total			2n+3 steps

The steps per execution of the statement is the amount by which the count changes as a result of execution of that statement. The frequency denotes the number of times each statement is executed.

Complexity of Algorithms

1. Best case: Inputs are given in such a way that minimum time is required to process them.
2. Average case: The amount of time the algorithm takes on an average set of inputs
3. Worst case: The amount of time the algorithm takes on the worst possible set of inputs.

Example: Linear Search



Example: Linear Search

	3	4	5	6	7	9	10	12	15
A	1	2	3	4	5	6	7	8	9

Best Case: If we want to search an element 3, whether it is present or not, first A[1] is compared with 3, and match is found. Thus total number of comparisons is only one. Such a search takes the minimum number or comparisons, so under best case,

Time complexity is O(1)

Average Case: If we want to search for the element 7, first A[1] is compared with 7, but no match occurs. Then A[2] is compared with 7 and once again, no match occurs. This is repeated until the 5th comparison, where A[5] is compared with 7 and here, match occurs. The number of comparisons is 5.

If there are n elements, then we require n/2 comparisons

$$\therefore \text{Time complexity is } O\left(\frac{n}{2}\right) = O(n) \text{ (we neglect constant)}$$

Worst Case: If we want to search for the element 15, then A[1] is compared to 15 first, and no match is found. This is repeated for the subsequent steps until the very last step, where A[9] is compared to 15.

Therefore time complexity is O(n)

Asymptotic Notation

Asymptotic complexity gives an idea of how rapidly the space or time requirement can grow as the problem size increases. There are four important asymptotic notations

1. Big oh (O) notation
2. Omega (Ω) notation
3. Theta (Θ) notation

Big oh notation - O

It is used to describe the efficiency of the algorithm, to represent the upper bound of an algorithm's running time.

Let f(n) and g(n) be the two non-negative functions. We say that f(n) is said to be O(g(n)) if and only if there exists a constant c and n_o such that

f(n) ≤ g(n) for all non-negative values of n where n ≥ n_0

Here g(n) is the upper bound for f(n)

Ex: Let $f(n) = 2n^4 + 5n2 + 2n + 3$

$\leq 2n^4 + 5n^4 + 2n^4 + 3n^4$

$\leq (2+5+2+3)n^4$

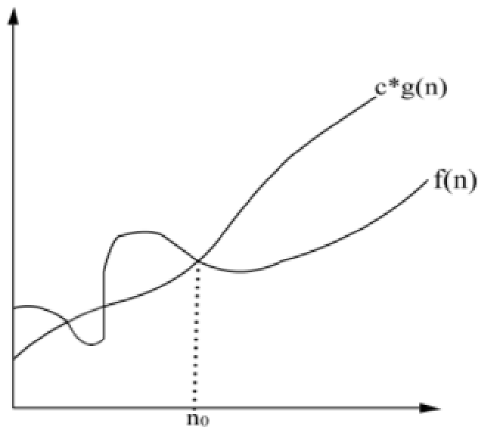
$\leq 12n^4.$

$\therefore f(n) = 12n^4$

This implies $g(n) = n^4,\ n \geq 1$

$\therefore c = 12$ and $n_0 = 1$

$\therefore f(n) = O(n^4)$



Ω Notation

It is used to represent the lower bound of an algorithm's running time.

The function f(n) = Ω(g(n)) if and only if there exists positive constants c and n_0 such that

f(n) ≥ c*g(n) for all n, n ≥ n_0

Example:

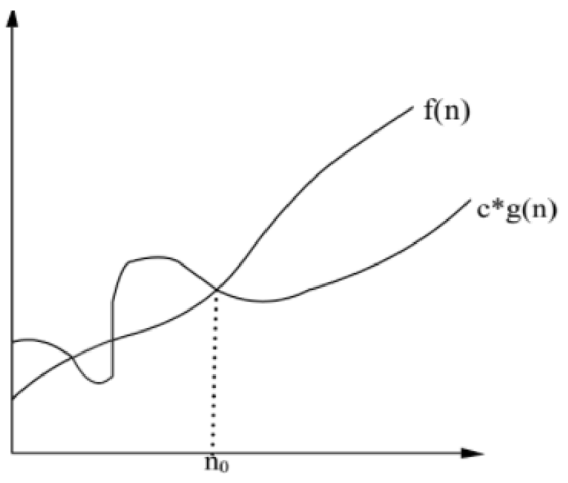
Let $f(n) = 2n^4 + 5n2 + 2n + 3$

$\geq 2n^4$ (for example as $n \rightarrow \infty$, lower order oterms are insignificant)

$\therefore f(n) \geq 2n^4, n \geq 1$

$\therefore g(n) = n^4$, $c=2$ and $n_0 = 1$

$\therefore f(n) = \Omega(n^4)$



Θ Notation

This notation is in between the upper and lower bound of an algorithm's running time.

Let f(n) and g(n) be two non-negative functions. We say that f(n) is said to be Θ(g(n)) if and only if there exists positive constants c_1 and c_2 such that:

c_1g(n) ≤ f(n) ≤ c_2g(n) for all non-negative values n where n ≥ n_0

This notation provides both upper and lower bounds of f(n), i.e. g(n) is both upper and lower bounds on the value of f(n), for large n. In other words the notation says that f(n) is both O(g(n)) and Ω(g(n)) for all n where n ≥ n_0

This function $f(n) = \theta(g(n))$ iff g(n) is both upper and lower bound an f(n).

Example:

$f(n) = 2n^4 + 5n^2 + 2n + 3$

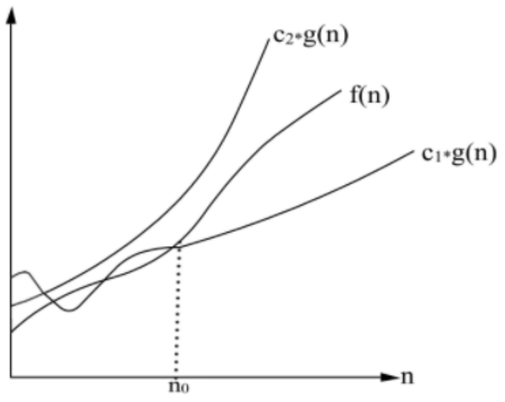
$\Rightarrow 2n^4 < 2n^4 + 5n^2 + 2n + 3 < 12n^4$

$\Rightarrow 2n^4 < f(n) < 12n^4$, n ⊳

$\therefore g(n) = n^4$

$\therefore c1=2, c2=12$ and $n0=1$

$\therefore f(n)=(n^4)$



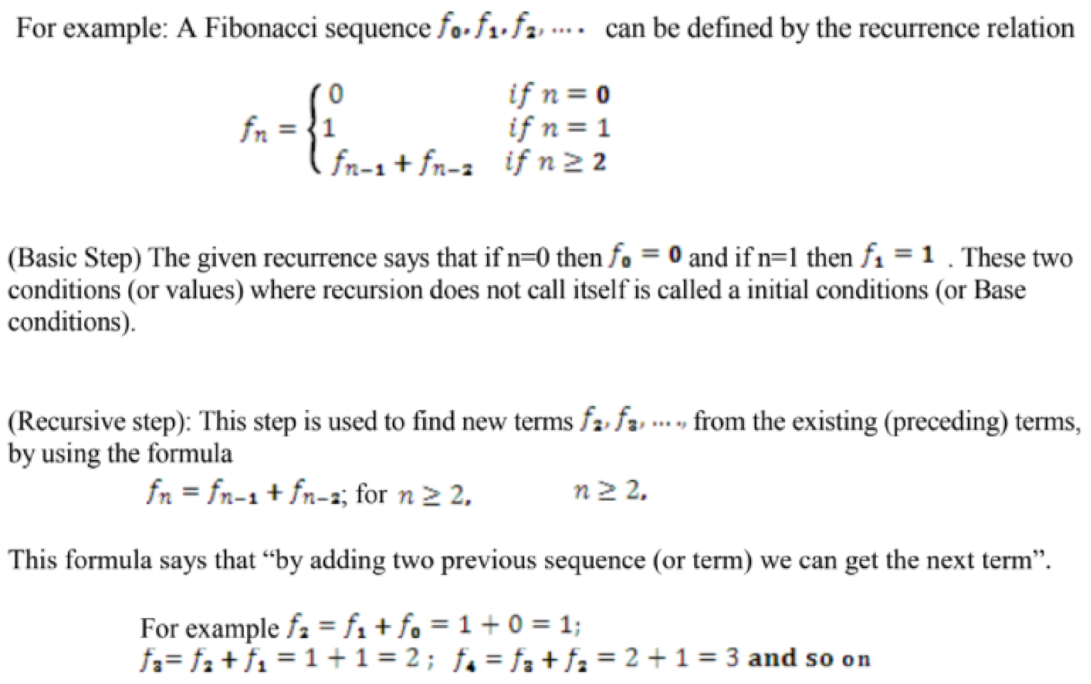
Recurrence relation

Recurrence relations often arise when calculating the time and space complexity of algorithms. We use it to describe the running time of recursive algorithms.

A recurrence relation is an equation or inequality that describes a function in terms of its value on smaller inputs or as a function of preceding terms. A recurrence consists of two steps:

1. Basic step. Here we have one or more constant values that are used to terminate recurrence. It is also known as initial or base conditions.
2. Recursive steps. This step is to find new terms from the preceding terms. Thus in this step the recurrence computes next sequence from the k preceding values f_{n-1}, f_{n-2}, . . . ,f_{n-k}

   This formula is known as a recurrence relation. This formula refers to itself, and the argument of the formula must be on the smaller values.



For example: A Fibonacci sequence $f_0, f_1, f_2, \ldots$ can be defined by the recurrence relation

$$f_n = \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f_{n-1} + f_{n-2} & if\ n \geq 2 \end{cases}$$

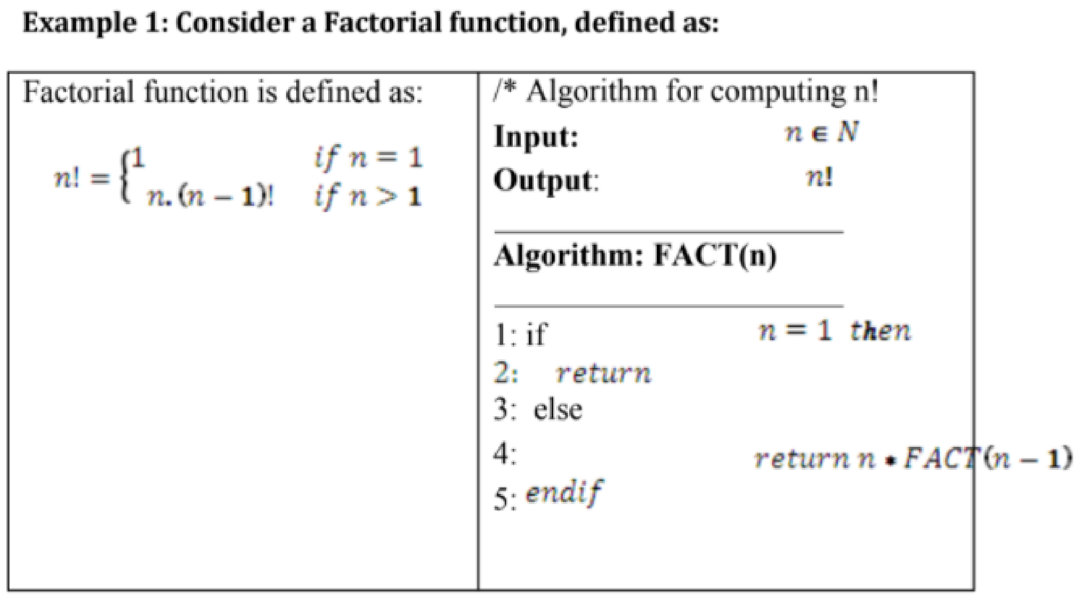
(Basic Step) The given recurrence says that if n=0 then $f_0 = 0$ and if n=1 then $f_1 = 1$. These two conditions (or values) where recursion does not call itself is called a initial conditions (or Base conditions).

(Recursive step): This step is used to find new terms $f_2, f_3, \ldots,$ from the existing (preceding) terms, by using the formula

$$f_n = f_{n-1} + f_{n-2};\ \text{for}\ n \geq 2, \qquad n \geq 2,$$

This formula says that "by adding two previous sequence (or term) we can get the next term".

For example $f_2 = f_1 + f_0 = 1 + 0 = 1;$
$f_3 = f_2 + f_1 = 1 + 1 = 2;\ \ f_4 = f_3 + f_2 = 2 + 1 = 3$ **and so on**



Example 1: Consider a Factorial function, defined as:

Factorial function is defined as:	/* Algorithm for computing n!
$n! = \begin{cases} 1 & if\ n = 1 \\ n.(n-1)! & if\ n > 1 \end{cases}$	**Input:** $n \in N$ **Output**: $n!$ **Algorithm: FACT(n)** 1: if $n = 1$ $then$ 2: $return$ 3: else 4: $return\ n * FACT(n-1)$ 5: $endif$

A recurrence relation can be solved using the following methods:

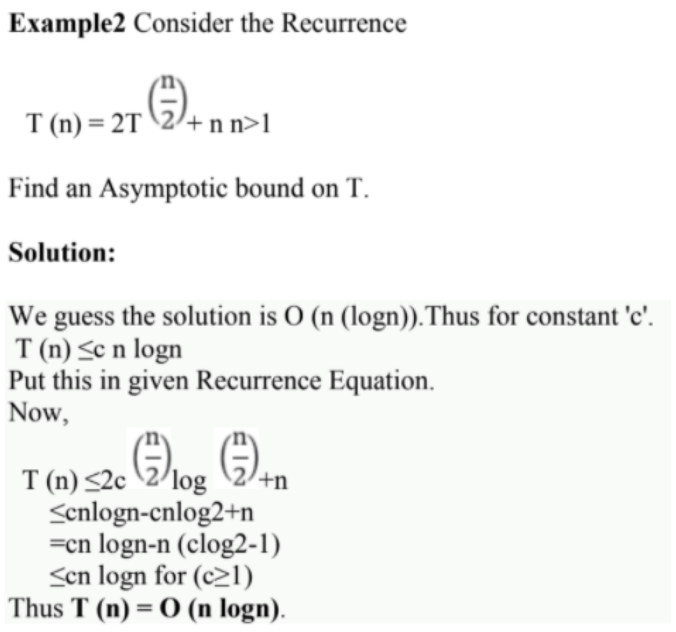
1. Substitution method

2. Iteration method
3. Recursion tree method
4. Master method

Substitution method

It consists of 2 main steps

1. Guess the solution
2. Use the mathematical induction to find the boundary condition and show that the guess is correct



Example2 Consider the Recurrence

$T(n) = 2T\left(\frac{n}{2}\right) + n \quad n>1$

Find an Asymptotic bound on T.

Solution:

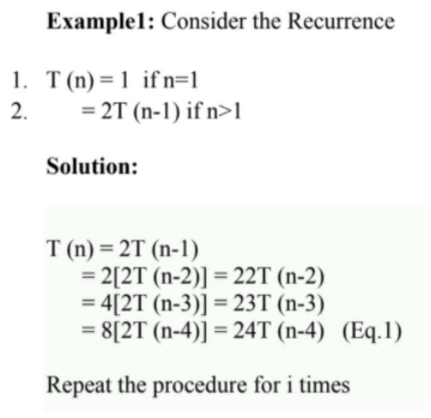
We guess the solution is O (n (logn)).Thus for constant 'c'.
$T(n) \leq c\, n \log n$
Put this in given Recurrence Equation.
Now,

$$\begin{aligned} T(n) &\leq 2c\left(\frac{n}{2}\right)\log\left(\frac{n}{2}\right)+n \\ &\leq cn\log n - cn\log 2 + n \\ &= cn\log n - n(c\log 2 - 1) \\ &\leq cn\log n \text{ for } (c \geq 1) \end{aligned}$$

Thus **T (n) = O (n logn)**.

Iteration method

It means to expand the recurrence and express it as a summation of terms and initial conditions.



Example1: Consider the Recurrence

1. T (n) = 1 if n=1
2. = 2T (n-1) if n>1

Solution:

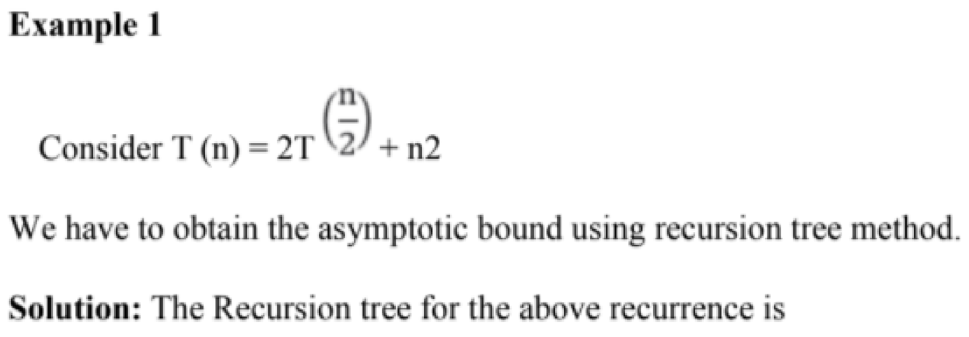
T (n) = 2T (n-1)
= 2[2T (n-2)] = 22T (n-2)
= 4[2T (n-3)] = 23T (n-3)
= 8[2T (n-4)] = 24T (n-4) (Eq.1)

Repeat the procedure for i times

Recursion tree method

It is a pictorial representation of iteration method which is in the form of a tree where at each level nodes are expanded. In general, we consider the second term in recurrence as root. It is useful where divide and conquer algorithm is used.

Each root and child represents the cost of a single subproblem. We sum the costs within each of the levels of the tree to obtain prelevel costs and then sum all of those to obtain the total cost of the recursion.

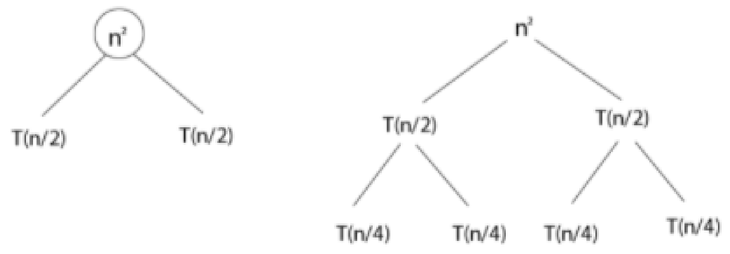


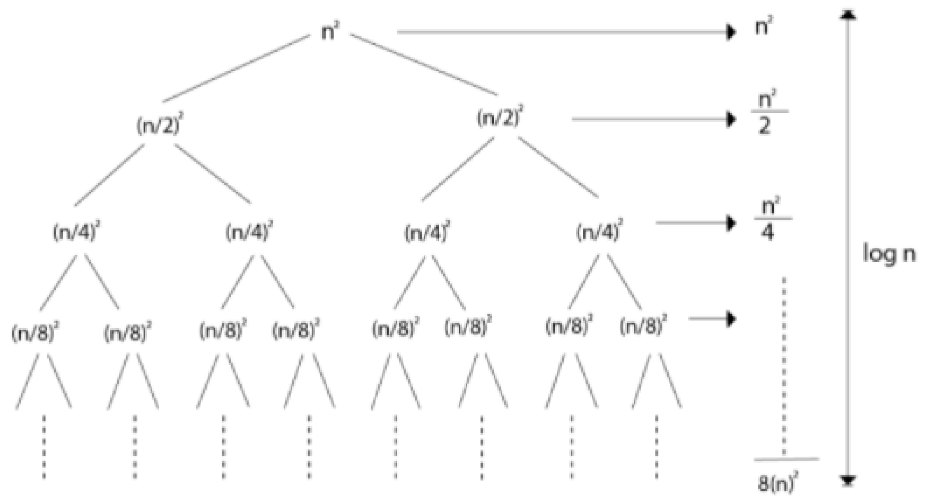
Example 1

Consider T (n) = 2T $\left(\frac{n}{2}\right)$ + n2

We have to obtain the asymptotic bound using recursion tree method.

Solution: The Recursion tree for the above recurrence is





$$T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \ldots\ldots \log n \text{ times.}$$

$$\leq n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2^i}\right)$$

$$\leq n^2 \left(\frac{1}{1-\frac{1}{2}}\right) \leq 2n^2$$

$$\mathbf{T\ (n) = \theta} n^2$$

Master Method

Applies for problems of the form:

aT($\frac{n}{b}$) + f(n), where a ≥1 and b >1

Solution will be of the form

$$T(n) = n^{log_b a}[U(n)]$$

U[n] depends on H[n]

H[n]	U[n]
n^r, r>0	O(n^r)

H[n]	U[n]
n^r, r<0	O(1)
(log_2n)^i, i≥0	$\frac{(log_2 n)^{i+1}}{i+1}$