UNIT I

INTRODUCTION TO ALGORITHM DESIGN

Introduction - Fundamentals of algorithm (Line count, operation count) - Algorithm Design Techniques (Approaches, Design Paradigms) - Designing an algorithm and its Analysis (Best, Worst & Average case) - Asymptotic Notations based on Orders of Growth - Mathematical Analysis - Induction - Recurrence Relation: Substitution method, Recursion method, Master's Theorem.

INTRODUCTION

Algorithm is a step by step procedure, which defines a set of instructions to be executed in certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

An algorithm is defined as follows:

- An algorithm is a set of rules for carrying out calculation either by hand or on a machine.
- An algorithm is a finite step-by-step procedure to achieve a required result.
- An algorithm is a sequence of computational steps that transform the input into the output.
- An algorithm is a sequence of operations performed on data that have to be organized in data structures.
- An algorithm is an abstraction of a program to be executed on a physical machine (model of Computation).

Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the below mentioned characteristics –

- Unambiguous – Algorithm should be clear and unambiguous. Each of its steps (or phases), and their input/outputs should be clear and must lead to only one meaning.
- Input – An algorithm should have 0 or more well defined inputs.
- Output – An algorithm should have 1 or more well defined outputs, and should match the desired output.
- Finiteness – Algorithms must terminate after a finite number of steps.
- Feasibility – Should be feasible with the available resources.
- Independent – An algorithm should have step-by-step directions which should be independent of any programming code.

ALGORITHM DESIGN TECHNIQUES (APPROACHES, DESIGN PARADIGMS)

General approaches to the construction of efficient solutions to problems. Such methods are of interest because:

- They provide templates suited to solving a broad range of diverse problems.
- They can be translated into common control and data structures provided by most high-level languages.
- The temporal and spatial requirements of the algorithms which result can be precisely analyzed.

Although more than one technique may be applicable to a specific problem, it is often the case that an algorithm constructed by one approach is clearly superior to equivalent solutions built using alternative techniques.

1. **Brute Force**

Brute force is a straightforward approach to solve a problem based on the problem's statement and definitions of the concepts involved. It is considered as one of the easiest approach to apply and is useful for solving small – size instances of a problem. Some examples of brute force algorithms are:

- Computing a^n ($a > 0$, n a nonnegative integer) by multiplying a*a*…*a
- Computing $n!$
- Selection sort , Bubble sort
- Sequential search
- Exhaustive search: Traveling Salesman Problem, Knapsack problem.

2. **Divide-and-Conquer, Decrease-and-Conquer**

These are methods of designing algorithms that (informally) proceed as follows:

Given an instance of the problem to be solved, split this into several smaller sub-instances (of the same problem), independently solve each of the sub-instances and then combine the sub-instance solutions so as to yield a solution for the original instance. With the divide-and-conquer method the size of the problem instance is reduced by a factor (e.g. half the input size), while with the decrease-and-conquer method the size is reduced by a constant.

Examples of divide-and-conquer algorithms:

- Computing an ($a > 0$, n a nonnegative integer) by recursion
- Binary search in a sorted array (recursion)
- Mergesort algorithm, Quicksort algorithm (recursion)
- The algorithm for solving the fake coin problem (recursion)

3. **Greedy Algorithms "take what you can get now" strategy**

The solution is constructed through a sequence of steps, each expanding a partially constructed solution obtained so far. At each step the choice must be locally optimal – this is the central point of this technique.

Greedy is a strategy that works well on optimization problems with the following characteristics:

1. **Greedy-choice property:** A global optimum can be arrived at by selecting a local optimum.
2. **Optimal substructure:** An optimal solution to the problem contains an optimal solution to subproblems.

The second property may make greedy algorithms look like dynamic programming. However, the two techniques are quite different.

Examples:

- Minimal spanning tree
- Shortest distance in graphs
- Greedy algorithm for the Knapsack problem
- The coin exchange problem
- Huffman trees for optimal encoding

Greedy techniques are mainly used to solve optimization problems. They do not always give the best solution.

Example:

Examples of such greedy algorithms are

- Kruskal's algorithm for finding minimum spanning trees
- Prim's algorithm for finding minimum spanning trees
- Huffman Algorithm for finding optimum Huffman trees.
- Used in Networking too

Greedy algorithms appear in network routing as well. Using greedy routing, a message is forwarded to the neighboring node which is "closest" to the destination. The notion of a node's location (and hence "closeness") may be determined by its physical location, as in geographic routing used by ad hoc networks. Location may also be an entirely artificial construct as in small world routing and distributed hash table

4. **Dynamic Programming**

One disadvantage of using Divide-and-Conquer is that the process of recursively solving separate sub-instances can result in the same computations being performed repeatedly since identical sub-instances may arise.

It is used when the solution can be recursively described in terms of solutions to subproblems (optimal substructure). Algorithm finds solutions to subproblems and stores them in memory for later use. More efficient than "brute-force methods", which solve the same subproblems over and over again.

- **Optimal substructure:**

Optimal solution to problem consists of optimal solutions to subproblems

- **Overlapping subproblems:**

Few subproblems in total, many recurring instances of each

- **Bottom up approach:**

Solve bottom-up, building a table of solved subproblems that are used to solve larger ones.

Examples:

- Fibonacci numbers computed by iteration.
- Warshall's algorithm implemented by iterations

5. **Backtracking methods**

The method is used for state-space search problems. State-space search problems are problems, where the problem representation consists of:

- initial state
- goal state(s)
- a set of intermediate states
- a set of operators that transform one state into another. Each operator has preconditions and post conditions.
- a cost function – evaluates the cost of the operations (optional)
- a utility function – evaluates how close is a given state to the goal state (optional)

The solving process solution is based on the construction of a state-space tree, whose nodes represent states, the root represents the initial state, and one or more leaves are goal states. Each edge is labeled with some operator.

> If a node b is obtained from a node a as a result of applying the operator O, then b is a child of a and the edge from a to b is labeled with O.

The solution is obtained by searching the tree until a goal state is found.

Backtracking uses depth-first search usually without cost function. The main algorithm is as follows:

1. Store the initial state in a stack
2. While the stack is not empty, do:
    a. Read a node from the stack.
    b. While there are available operators do:
        i. Apply an operator to generate a child
        ii. If the child is a goal state – stop
        iii. If it is a new state, push the child into the stack

The utility function is used to tell how close is a given state to the goal state and whether a given state may be considered a goal state.

If no children can be generated from a given node, then we backtrack – read the next node from the stack.

6. **Branch-and-bound**

Branch and bound is used when we can evaluate each node using the cost and utility functions. At each step we choose the best node to proceed further. Branch-and bound algorithms are implemented using a priority queue. The state-space tree is built in a **breadth-first** manner.

Example: the 8-puzzle problem. The cost function is the number of moves. The utility function evaluates how close is a given state of the puzzle to the goal state, e.g. counting how many tiles are not in place.

ALGORITHM ANALYSIS

An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties:

1. **Time Complexity**
2. **Space Complexity**

Suppose X is an algorithm and n is the size of input data, the time and space used by the Algorithm X are the two main factors which decide the efficiency of X.

- **Time Factor –** The time is measured by counting the number of key operations such as comparisons in sorting algorithm
- **Space Factor –** The space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm f(n) gives the running time and / or storage space required by the algorithm in terms of n as the size of input data.

Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. Its the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

Space required by an algorithm is equal to the sum of the following two components −

- **A fixed part** that is a space required to store certain data and variables that are independent of the size of the problem. For example, simple variables & constant used and program size etc.
- **A variable part** is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stacks space etc.

An algorithm generally requires space for following components:

- **Instruction Space:** It is the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.
- **Data Space:** It is the space required to store all the constants and variables value.
- **Environment Space:** It is the space required to store the environment information needed to resume the suspended function.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I) Where C is the fixed part and S(I) is the variable part of the algorithm which depends on instance characteristic I.

Time Complexity

The time complexity is a function that gives the amount of time required by an algorithm to run to completion.

- Worst case time complexity: It is the function defined by the maximum amount of time needed by an algorithm for an input of size n.
- Average case time complexity: The average-case running time of an algorithm is an estimate of the running time for an “average” input. Computation of average-case running time entails knowing all possible input sequences, the probability distribution of occurrence of these sequences, and the running times for the individual sequences.
- Best case time complexity: It is the minimum amount of time that an algorithm requires for an input of size n.

There are four rules to count the operations:

Rule 1: for loops - the size of the loop times the running time of the body

The running time of a for loop is at most the running time of the statements inside the loop times the number of iterations.

```
for( i = 0; i < n; i++)

sum = sum + i;
```

a. Find the running time of statements when executed only once:

The statements in the loop heading have fixed number of operations, hence they have constant running time O(1) when executed only once. The statement in the loop body has fixed number of operations, hence it has a constant running time when executed only once.

b. Find how many times each statement is executed.

```
for( i = 0; i < n; i++) // i = 0; executed only once: O(1)

// i < n; n + 1 times O(n)

// i++ n times O(n)

// total time of the loop heading:

// O(1) + O(n) + O(n) = O(n)

sum = sum + i; // executed n times, O(n)
```

The loop heading plus the loop body will give: O(n) + O(n) = O(n).

Loop running time is: O(n)

Mathematical analysis of how many times the statements in the body are executed

$$C(n) = \sum_{i=0}^{n-1} 1 = (n-1) - 0 + 1 = n$$

If

a) the size of the loop is n (loop variable runs from 0, or some fixed constant, to n) and
b) the body has constant running time (no nested loops)

then the time is O(n)

Rule 2: Nested loops – the product of the size of the loops times the running time of the body

The total running time is the running time of the inside statements times the product of the sizes of all the loops

```
sum = 0;

for( i = 0; i < n; i++)

        for( j = 0; j < n; j++)

                sum++;
```

Applying Rule 1 for the nested loop (the 'j' loop) we get O(n) for the body of the outer loop. The outer loop runs n times, therefore the total time for the nested loops will be

O(n) * O(n) = O(n*n) = O(n^2)

Analysis

Inner loop:

$$S(i) = \sum_{j=0}^{n-1} 1 = (n-1) - 0 + 1 = n$$

Outer loop:

$$C(n) = \sum_{i=0}^{n-1} S(i) = \sum_{i=0}^{n-1} n = n * \sum_{i=0}^{n-1} 1 = n((n-1) - 0 + 1) = n^2$$

Inner loop:

$$S(i) = \sum_{j=0}^{n-1} 1 = (n-1) - 0 + 1 = n$$

Outer loop:

$$C(n) = \sum_{i=0}^{n-1} S(i) = \sum_{i=0}^{n-1} n = n * \sum_{i=0}^{n-1} 1 = n((n-1) - 0 + 1) = n^2$$

What happens if the inner loop does not start from 0?

```
sum = 0;

for( i = 0; i < n; i++)

        for( j = i; j < n; j++)

            sum++;
```

Here, the number of the times the inner loop is executed depends on the value of i

i = 0, inner loop runs n times

i = 1, inner loop runs (n-1) times

i = 2, inner loop runs (n-2) times

…

i = n – 2, inner loop runs 2 times

i = n – 1, inner loop runs once.

Thus we get: (1 + 2 + … + n) = $n*(n+1)/2 = O(n^2)$

General rule for nested loops:

Running time is the product of the size of the loops times the running time of the body.

Example:

```
sum = 0;
for( i = 0; i < n; i++)
        for( j = 0; j < 2n; j++)
                sum++;
```

We have one operation inside the loops, and the product of the sizes is $2n^2$

Hence the running time is $O(2n^2) = O(n^2)$

Note: if the body contains a function call, its running time has to be taken into consideration

```
sum = 0;
for( i = 0; i < n; i++)
        for( j = 0; j < n; j++)
                sum = sum + function(sum);
```

Assume that the running time of function(sum) is known to be log(n).

Then the total running time will be $O(n^2*\log(n))$

Rule 3: Consecutive program fragments

The total running time is the maximum of the running time of the individual fragments

```
sum = 0;
        for( i = 0; i < n; i++)
                sum = sum + i;
sum = 0;
for( i = 0; i < n; i++)
        for( j = 0; j < 2*n; j++)
                sum++;
```

The first loop runs in O(n) time, the second - O(n2) time, the maximum is O(n2)

Rule 4: If statement

```
if C
        S1;
else
        S2;
```

The running time is the maximum of the running times of S1 and S2.

Summary

Steps in analysis of non-recursive algorithms:

- Decide on parameter n indicating input size
- Identify algorithm's basic operation
- Check whether the number of time the basic operation is executed depends on some additional property of the input. If so, determine worst, average, and best case for input of size n
- Count the number of operations using the rules above.

ASYMPTOTIC NOTATIONS

The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, and doesn't require algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.

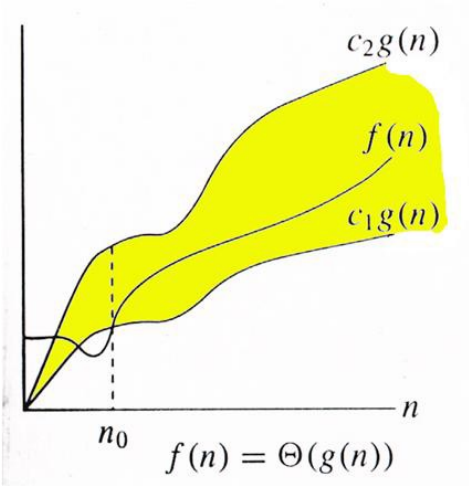
1) Θ Notation:

The theta notation bounds a function from above and below, so it defines exact asymptotic behavior. A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression. $3n^3 + 6n^2 + 6000 = \Theta(n^3)$

Dropping lower order terms is always fine because there will always be a n0 after which $\Theta(n^3)$ beats $\Theta(n^2)$ irrespective of the constants involved. For a given function g(n), we denote Θ(g(n)) is following set of functions.

$$\Theta((g(n)) = \{f(n): \textit{there exist positive constants } c1, c2 \textit{ and } n0 \textit{ such that}$$

$$0 <= c1 * g(n) <= f(n) <= c2 * g(n) \textit{ for all } n >= n0\}$$



The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.

2. Big O Notation:

The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time. If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:

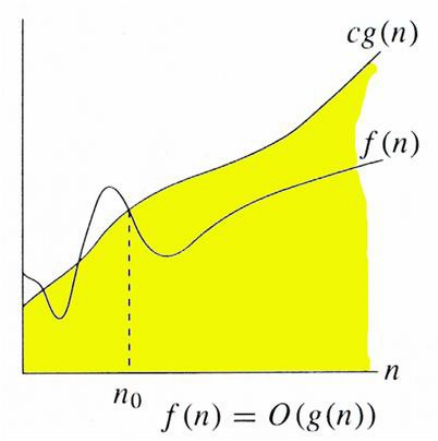
1. The worst case time complexity of Insertion Sort is Θ(n^2).

2. The best case time complexity of Insertion Sort is Θ(n).

The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

$$O(g(n)) = \{ f(n): \textit{there exist positive constants } c \textit{ and } n0 \textit{ such that}$$

$$0 <= f(n) <= cg(n) \textit{ for all } n >= n0\}$$

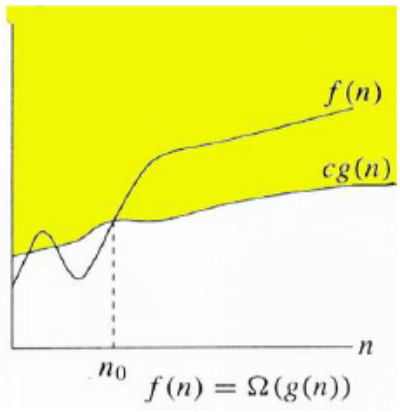


3) Ω Notation:

Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound. Ω Notation< can be useful when we have lower bound on time complexity of an algorithm. As discussed in the previous post, the best case performance of an algorithm is generally not useful; the Omega notation is the least used notation among all three.

For a given function g(n), we denote by Ω(g(n)) the set of functions.

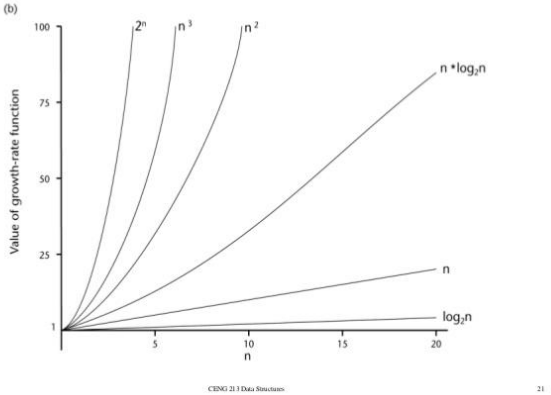
$$\Omega\ (g(n))\ =\ \{f(n)\text{: there exist positive constants } c \text{ and } n0 \text{ such that}$$
$$0\ <=\ cg(n)\ <=\ f(n) \text{ for all } n\ >=\ n0\}.$$



Rate of Growth functions

	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	O(1)	O(log *n*)	O(*n*)	O(*n* log *n*)	O(n^2)	O(n^3)	O(2^n)
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

A Comparison of Growth-Rate Functions (cont.)



Linear and Binary search

An algorithm is a step-by-step procedure or method for solving a problem by a computer in a given number of steps. The steps of an algorithm may include repetition depending upon the problem for which the algorithm is being developed. The algorithm is written in human readable and understandable form. To search an element in a given array, it can be done in two ways linear search and Binary search.

Linear Search

A linear search is the basic and simple search algorithm. A linear search searches an element or value from an array till the desired element or value is not found and it searches in a sequence order. It compares the element with all the other elements given in the list and if the element is matched it returns the value index else it return -1. Linear Search is applied on the unsorted or unordered list when there are fewer elements in a list.

Pseudocode:-

```
# Input: Array D, integer key
# Output: first index of key in D, or -1 if not found
 For i = 0 to last index of D:
  if D[i] == key:
    return i
 return -1
```

Example with Implementation

To search the element 5 it will go step by step in a sequence order.

```
linear(a[n], key)
   for( i = 0; i < n; i++)
        if (a[i] == key)
            return i;
  return -1;
```

Asymptotic Analysis

Worst Case Analysis (Usually Done)

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (target in the above code) is not present in the array. When target is not present, the search() functions compares it with all the elements of array one by one. Therefore, the worst case time complexity of linear search would be $\Theta(n)$.

Average Case Analysis (Sometimes done)

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of target not being present in array).

The key is equally likely to be in any position in the array

If the key is in the first array position: 1 comparison

If the key is in the second array position: 2 comparisons

...

If the key is in the ith postion: i comparisons

...

So average all these possibilities: (1+2+3+...+n)/n = [n(n+1)/2] /n = (n+1)/2 comparisons. The average number of comparisons is **(n+1)/2 = Θ(n).**

Best Case Analysis (Bogus)

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when Target is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be Θ(1)

Binary Search

Binary Search is applied on the sorted array or list. In binary search, we first compare the value with the elements in the middle position of the array. If the value is matched, then we return the value. If the value is less than the middle element, then it must lie in the lower half of the array and if it's greater than the element then it must lie in the upper half of the array. We repeat this procedure on the lower (or upper) half of the array. Binary Search is useful when there are large numbers of elements in an array.

```
binarysearch(a[n], key, low, high)
while(low<high)
{
mid = (low+high)/2;
if(a[mid]=key)
        return mid;
elseif (a[mid] > key)
        high=mid-1;
 else
        low=mid+1;
 }
return -1;
```

In the above program logic, we are first comparing the middle number of the list, with the target, if it matches we return. If it doesn't, we see whether the middle number is greater than or smaller than the target.

If the Middle number is greater than the Target, we start the binary search again, but this time on the left half of the list, that is from the start of the list to the middle, not beyond that.

If the Middle number is smaller than the Target, we start the binary search again, but on the right half of the list, that is from the middle of the list to the end of the list.

Complexity Analysis

Worst case analysis: The key is not in the array

Let T(n) be the number of comparisons done in the worst case for an array of size n. For the purposes of analysis, assume n is a power of 2, ie n = 2^k.

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

$= 2 + 2 + T(\frac{n}{2^2})$ // 2nd iteration

$= 2 + 2 + 2 + T(n/2^3)$ // 3rd iteration

...

$= i * 2 + T(n/2^i)$ // ith iteration

... $= k * 2 + T(1)$

Note that k = logn, and that T(1) = 2.

So T(n) = 2logn + 2 = O(logn)

So we expect binary search to be significantly more efficient than linear search for large values of n.

Bubble Sort and Insertion Sort

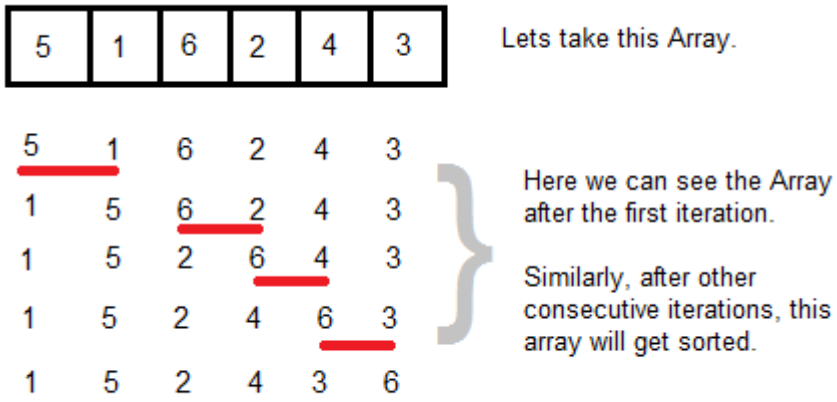
Bubble Sort

Bubble Sort is an algorithm which is used to sort N elements that are given in a memory for eg: an Array with N number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

It is called Bubble sort, because with each iteration the smaller element in the list bubbles up towards the first place, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.

Bubble Sort for Data Structures



Sorting using Bubble Sort Algorithm

Let's consider an array with values {5, 1, 6, 2, 4, 3}

```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, temp;
for(i=0; i<6; i++)
{
 for(j=0; j<6-i-1; j++)
 {
  if( a[j] > a[j+1])
  {
   temp = a[j];
   a[j] = a[j+1];
   a[j+1] = temp;
  }
 }
}
```

//now you can print the sorted array after this

Above is the algorithm, to sort an array using Bubble Sort. Although the above logic will sort and unsorted array, still the above algorithm isn't efficient and can be enhanced further. Because as per the above logic, the for loop will keep going for six iterations even if the array gets sorted after the second iteration.

Hence we can insert a flag and can keep checking whether swapping of elements is taking place or not. If no swapping is taking place that means the array is sorted and wew can jump out of the for loop.

```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, temp;
for(i=0; i<6; i++)
{
 int flag = 0;       //taking a flag variable
 for(j=0; j<6-i-1; j++)
 {
  if( a[j] > a[j+1])
  {
   temp = a[j];
   a[j] = a[j+1];
   a[j+1] = temp;
   flag = 1;       //setting flag as 1, if swapping occurs
  }
 }
 if(!flag)          //breaking out of for loop if no swapping takes place
 {
  break;
 }
}
```

In the above code, if in a complete single cycle of j iteration(inner for loop), no swapping takes place, and flag remains 0, then we will break out of the for loops, because the array has already been sorted.

Complexity Analysis of Bubble Sorting

In Bubble Sort, n-1 comparisons will be done in 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be

$(n-1)+(n-2)+(n-3)+\ldots..+3+2+1$

$Sum = n(n-1)/2$

$i.e\ O(n2)$

Hence the complexity of Bubble Sort is **$O(n^2)$**.

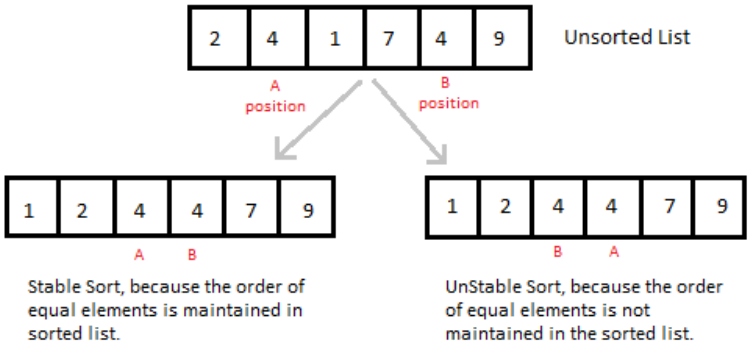
The main advantage of Bubble Sort is the simplicity of the algorithm. Space complexity for Bubble Sort is **O(1)**, because only single additional memory space is required for **temp** variable

Best-case Time Complexity will be **O(n)**, it is when the list is already sorted.

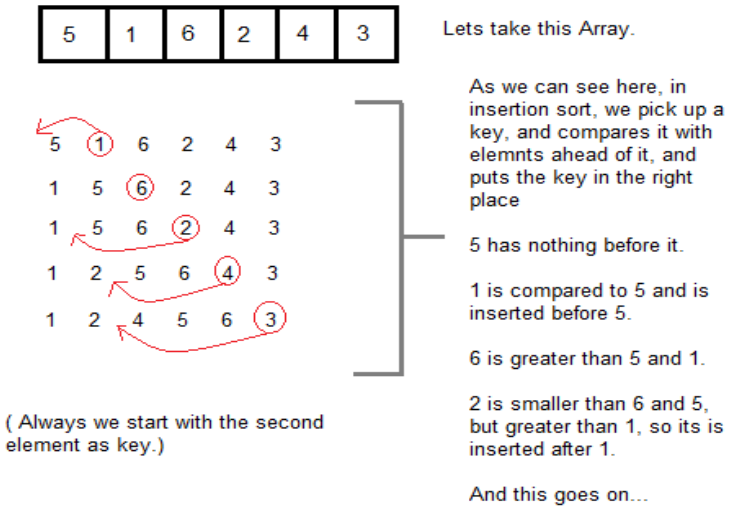
Insertion Sorting

It is a simple Sorting algorithm which sorts the array by shifting elements one by one. Following are some of the important characteristics of Insertion Sort.

1. It has one of the simplest implementation
2. It is efficient for smaller data sets, but very inefficient for larger lists.
3. Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
4. It is better than Selection Sort and Bubble Sort algorithms.
5. Its space complexity is less, like Bubble Sorting, inerstion sort also requires a single additional memory space.
6. It is Stable, as it does not change the relative order of elements with equal keys



How Insertion Sorting Works



Sorting using Insertion Sort Algorithm

```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, key;
for(i=1; i<6; i++)
{
 key = a[i];
 j = i-1;
 while(j>=0 && key < a[j])
 {
  a[j+1] = a[j];
  j--;
 }
 a[j+1] = key;
}
```

Now lets, understand the above simple insertion sort algorithm. We took an array with 6 integers. We took a variable key, in which we put each element of the array, in each pass, starting from the second element, that is a[1].

Then using the while loop, we iterate, until j becomes equal to zero or we find an element which is greater than key, and then we insert the key at that position.

In the above array, first we pick 1 as key, we compare it with 5(element before 1), 1 is smaller than 5, we shift 1 before 5. Then we pick 6, and compare it with 5 and 1, no shifting this time. Then 2 becomes the key and is compared with, 6 and 5, and then 2 is placed after 1. And this goes on, until complete array gets sorted.

Complexity Analysis of Insertion Sorting

- Worst Case Time Complexity : O(n^2)
- Best Case Time Complexity : O(n)
- Average Time Complexity : O(n^2)
- Space Complexity : O(1)

Sorting Algorithms	Best Case	Average Case	Worst Case
Quick Sort	O(n log(n))	O(n log(n))	$O(n^2)$
Merge Sort	O(n log(n))	O(n log(n))	O(n log(n))
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$
Heap Sort	O(n log(n))	O(n log(n))	O(n log(n))

MATHEMATICAL ANALYSIS

A recurrence is a recursive description of a function, or in other words, a description of a function in terms of itself. Like all recursive structures, a recurrence consists of one or more base cases and one or more recursive cases. Each of these cases is an equation or inequality, with some function value $f\ (n)$ on the left side. The base cases give explicit values for a (typically finite, typically small) subset of the possible values of n. The recursive cases relate the function value $f\ (n)$ to function value $f\ (k)$ for one or more integers $k\ <\ n$; typically, each recursive case applies to an infinite number of possible values of n.

For example, the following recurrence (written in two different but standard ways) describes the identity function $f\ (n)\ =\ n$:

$$f\ (n)\ = \begin{cases} 0 & if\ n\ =\ 0 \\ f(n\ -\ 1)\ +\ 1 & otherwise \end{cases} \qquad \begin{aligned} f\ (0)\ &=\ 0 \\ f\ (n)\ &=\ f\ (n-1)\ +\ 1 \quad for\ all\ n\ >\ 0 \end{aligned}$$

In both presentations, the first line is the only base case, and the second line is the only recursive case. The same function can satisfy many different recurrences; for example, both of the following recurrences also describe the identity function:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) & \text{otherwise} \end{cases} \qquad f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2 \cdot f(n/2) & \text{if } n \text{ is even and } n > 0 \\ f(n-1) + 1 & \text{if } n \text{ is odd} \end{cases}$$

We say that a particular function satisfies a recurrence, or is the solution to a recurrence, if each of the statements in the recurrence is true. Most recurrences—at least, those that we will encounter in this class—have a solution; moreover, if every case of the recurrence is an equation, that solution is unique. Specifically, if we transform the recursive formula into a recursive algorithm, the solution to the recurrence is the function computed by that algorithm!

Mathematical Analysis – Induction

Consider a recursive algorithm to compute the maximum element in an array of integers. You may assume the existence of a function "$max(a, b)$ " that returns the maximum of two integers a and b .

Algorithm: Finding the maximum in an array of n elements

```
Function FIND − ARRAY − MAX (A , n )
1: if (n = 1 ) then
2:         return (A[1] )
3: else
4:         return (max (A[n] , FIND − ARRAY − MAX (A , n − 1 ) ))
5: end if
```

Solution: We first need to formulate the proposition for algorithm correctness. In this case, we let $\boldsymbol{P(n)}$ stand for the proposition that Algorithm finds and returns the maximum integer in the locations $A[\mathbf{1}]$ through $A[\boldsymbol{n}]$. Accordingly, we have to show that $(\forall \boldsymbol{n})\ \boldsymbol{P(n)}$ is true.

BASIS: When there is only one element in the array , i.e., $\boldsymbol{n = 1}$, then this element is clearly the maximum element and it is returned on Line **2**. We thus see that $\boldsymbol{P(1)}$ is true.

INDUCTIVE STEP: Assume that Algorithm finds and returns the maximum element, when there are exactly **k** elements in **A**.

Now consider the case in which there are **k + 1** elements in **A**. Since **(k + 1) > 1**, Line **4** will be executed. In this step, we first make a recursive call to FIND-ARRAY-MAX with exactly **k** elements. From the inductive hypothesis, we know that the maximum elements in A**[1]** through A**[k]** is returned. Now the maximum element in **A** is either $A[\boldsymbol{k+1}]$ or the maximum element in $A[\mathbf{1}]\ through\ A[\boldsymbol{k}]\ (say\ \boldsymbol{r})$. Thus, returning the maximum of $A[\boldsymbol{k+1}]\ and\ \boldsymbol{r}$ clearly gives the maximum element in A, thereby proving that $\boldsymbol{P(k) \rightarrow P(k+1)}$. By applying the first principle of mathematical induction, we can conclude that $(\forall \boldsymbol{n})\boldsymbol{P(n)}$is true, i.e., Algorithm is correct.

Example

Find the exact solution to the recurrence relation using mathematical induction method

$$T(1) = 0$$

$$T(n) = 2\,T\left(\frac{n}{2}\right) + n, \qquad n \geq 2$$

$is\ T(n) = n\ log\ n$.

Solution:

Basis: At **n = 1**, both the closed form and the recurrence relation agree ($\mathbf{0 = 0}$) and so the basis is true.

Inductive step: Assume that $\boldsymbol{T(r) = r\ log\ r}$ for all $\mathbf{1} \leq \boldsymbol{r} \leq \boldsymbol{k}$. Now consider $\boldsymbol{T(k+1)}$. As per the recurrence relation, we have,

$$T(k+1) = 2\,T(\frac{k+1}{2}) + (k+1);\ since\ (k+1) \geq 2$$

$$= 2\left(\frac{(k+1)}{2}\ log\frac{k+1}{2}\right) + (k+1) as\ per\ the\ inductive\ hypothesis;\ since\ \frac{k+1}{2} < k$$

$$= (k+1)\,[log(k+1) - log\ 2] + (k+1)$$

$$= (k+1) log(k+1) - (k+1) + (k+1)$$

$$= (k+1)\ log(k+1)$$

We can therefore apply the second principle of mathematical induction to conclude that the exact solution to the given recurrence is $\boldsymbol{n\ log\ n}$.

Recurrence relation

Observe that the function $max(a, b)$ uses exactly one comparison. Thus, the comparison complexity of Algorithm can be described the recurrence relation:

$T(1) = 0$

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

This recurrence can be expanded as $\boldsymbol{T(n) = 1 + 1 + \cdots 1}$ $\boldsymbol{(n-1)}$ times to give $\boldsymbol{T(n) = n - 1}$

Many algorithms are recursive in nature. When we analyze them, we get a recurrence relation for time complexity. We get running time on an input of size n as a function of n and the running time on inputs of smaller sizes.

For example in Merge Sort, to sort a given array, we divide it in two halves and recursively repeat the process for the two halves. Finally we merge the results. Time complexity of Merge Sort can be written as T(n) = 2T(n/2) + cn. There are many other algorithms like Binary Search, Tower of Hanoi, etc.

There are mainly three ways for solving recurrences.

1) **Substitution Method**:

The substitution method for solving recurrences consists of two steps:

1) Guess the form of the solution.

2) Use mathematical induction to find constants in the form and show that the solution works..

For example consider the recurrence T(n) = 2T(n/2) + n

We guess the solution as T(n) = O(nLogn). Now we use induction to prove our guess.

We need to prove that T(n) <= cnLogn. We can assume that it is true for values smaller than n.

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

<= cn/2Log(n/2) + n

<= cnLogn - cnLog2 + n

<= cnLogn - cn + n

<= cnLogn

3) **Recurrence Tree Method:**

Many divide and conquer algorithms give us running-time recurrences of the form

$$\boldsymbol{T(n) = a\,T(n/b) + f(n)}$$

where a and b are constants and f **(**n**)** is some other function. There is a simple and general technique for solving many recurrences in this and similar forms, using a recursion tree. The root of the recursion tree is a box containing the value f **(**n**)**; the root has a children, each of which is the root of a (recursively defined) recursion tree for the function T**(**n/b**)**.

Equivalently, a recursion tree is a complete *a*-ary tree where each node at depth i contains the value f **(**n/b^i**)**. The recursion stops when we get to the base case(s) of the recurrence. Because we're only looking for asymptotic bounds, the exact base case doesn't matter; we can

safely assume that $T(\mathbf{1}) = \mathbf{\Theta}(\mathbf{1})$, or even that T**(**n**) = Θ (1)** for all n **≤** $\mathbf{10^{100}}$. I'll also assume for

simplicity that n is an integral power of b; we'll see how to avoid this assumption later (but to

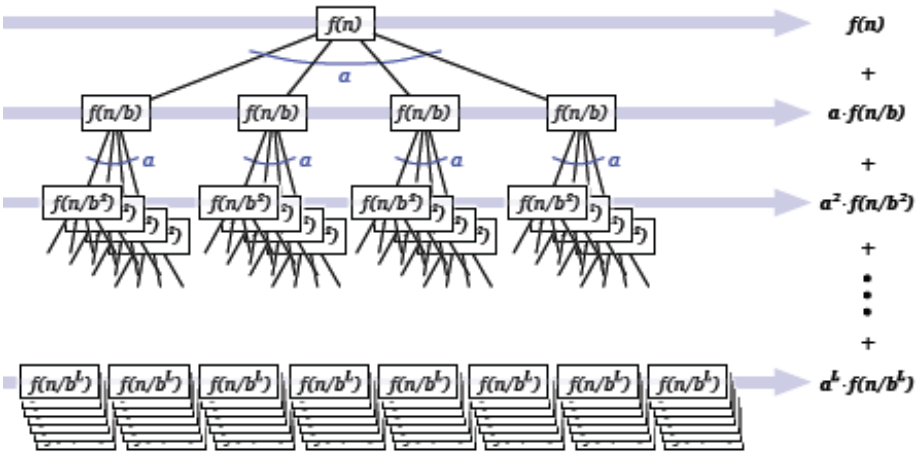
summarize: it doesn't matter).

Now T**(**n**)** is just the sum of all values stored in the recursion tree. For each i, the ith level of the tree contains a^i nodes, each with value f **(**n/b^i**)**. Thus,

$$T(n) = \sum_{i=0}^{L} a^i f(n/b^i)$$

where L is the depth of the recursion tree. We easily see that $L = \log_b n$, because $n/b^L = \mathbf{1}$. The base case $f(\mathbf{1}) = \boldsymbol{\Theta}(\mathbf{1})$ implies that the last non-zero term in the summation is **Θ** (aL) = **Θ** ($a^{\log_b n}$) = **Θ** ($n^{\log_b a}$).

For most divide-and-conquer recurrences, the level-by-level sum (∑) is a geometric series— each term is a constant factor larger or smaller than the previous term. In this case, only the largest term in the geometric series matters; all of the other terms are swallowed up by the **Θ** (.) notation.



A recursion tree for the recurrence $T(n) = a\,T(n/b) + f(n)$

Mergesort (simplified): $T(n) = 2T(n/2) + n$

There are 2^i nodes at level i, each with value $n/2^i$, so every term in the level-by-level sum (Σ) is the same:

$$T(n) = \sum_{i=0}^{L} n.$$

The recursion tree has $L = \log_2 n$ levels, so $T(n) = \Theta(n \log n)$.

In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically an arithmetic or geometric series. For example consider the recurrence relation .

For example

```
T(n) = T(n/4) + T(n/2) + cn^2

            cn^2
           /    \
      T(n/4)    T(n/2)

If we further break down the expression T(n/4) and T(n/2), we get following
recursion tree.

                cn^2
              /      \
         c(n^2)/16    c(n^2)/4
        /      \        /     \
  T(n/16)    T(n/8)  T(n/8)   T(n/4)

Breaking down further gives us following

                  cn^2
              /          \
         c(n^2)/16      c(n^2)/4
        /      \        /     \
c(n^2)/256  c(n^2)/64  c(n^2)/64  c(n^2)/16
 /    \      /    \    /    \      /    \
```

To know the value of T(n), we need to calculate sum of tree nodes level by level. If we sum the above tree level by level, we get the following series

T(n) = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ...

The above series is geometrical progression with ratio 5/16. To get an upper bound, we can sum the infinite series. We get the sum as $(n^2)/(1 - 5/16)$ which is $O(n^2)$

3) Master Method:

Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type.

$T(n) = aT(n/b) + f(n)$ where $a >= 1$ and $b > 1$

There are following three cases:

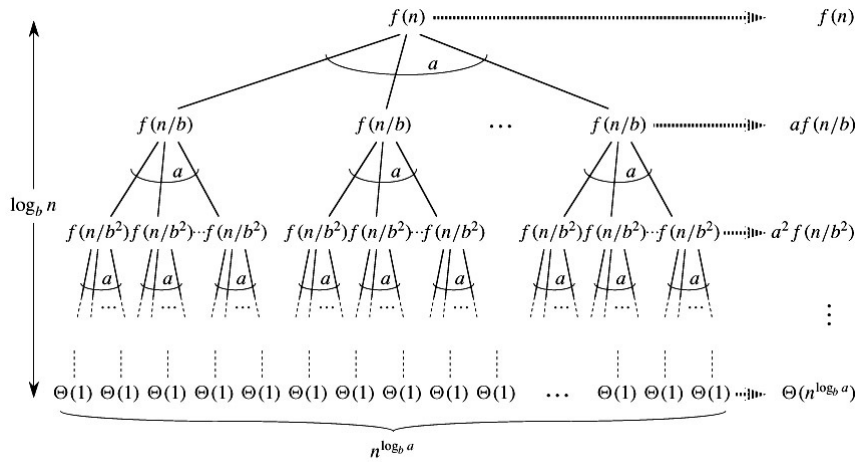
1.	If $f(n) = \Theta(n^c)$ where $c < \text{Log}_b a$ then $T(n) = \Theta(n^{\text{Log}_b a})$
2.	If $f(n) = \Theta(n^c)$ where $c = \text{Log}_b a$ then $T(n) = \Theta(n^c \text{Log } n)$
3.	If $f(n) = \Theta(n^c)$ where $c > \text{Log}_b a$ then $T(n) = \Theta(f(n))$

How does this work?

Master method is mainly derived from recurrence tree method. If we draw recurrence tree of

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

we can see that the work done at root is $f(n)$ and work done at all leaves is $\Theta(n^c)$ where c is $\text{Log}_b a$. And the height of recurrence tree is $\text{Log}_b n$



In recurrence tree method, calculate total work done. If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1). If work done at leaves and root is asymptotically same, then our result becomes height multiplied by work done at any level (Case 2). If work done at root is asymptotically more, then our result becomes work done at root (Case 3).

Examples

Merge Sort: $T(n) = 2T(n/2) + \Theta(n)$. It falls in case 2 as c is 1 and $\text{Log}_b a]$ is also 1. So the solution is $\Theta(n \text{ Logn})$

Binary Search: $T(n) = T(n/2) + \Theta(1)$. It also falls in case 2 as c is 0 and $\text{Log}_b a$ is also 0. Hence solution is $\Theta(\text{Logn})$.