

1.1 Fundamentals of Algorithm

Algorithm:

- Step by step procedure to solve a computational problem is called Algorithm.

or

- An Algorithm is a step-by-step plan for a computational procedure that possibly begins with an input and yields an output value in a finite number of steps in order to solve a particular problem.

Properties of Algorithm:

To Evaluate An Algorithm We Have To Satisfy The Following Criteria:

1. **Input:** The Algorithm should be given zero or more input.
 2. **Output:** At least one quantity is produced. For each input the algorithm produced value from specific task.
 3. **Definiteness:** Each instruction is clear and unambiguous.
 4. **Finiteness:** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
 5. **Effectiveness:** Every instruction must very basic so that it can be carried out, in principle, by a person using only pencil & paper.
- **Algorithm:** A well-defined computational procedure that takes some value, or set of values, as *input* and produces some value, or set of values, as *output*.
 - **Pseudocode:** A notation resembling a simplified programming language, used in program design.
 - **Program:** Collection of instructions which are ready to execute.

How to write an Algorithm:

There are no certain rules to write an algorithm. We can follow any strategy to design an algorithm which should be written in a step by step process.

Step 1: begin

Step 2: read a, b, c

Step 3: if $a > b$

 if $a > c$

 print a is largest

 else

 if $b > c$

 print b is largest

 else

 print c is largest

step 4: end

OR

Step 1: begin

Step 2: read a, b, c

Step 3: if $a > b$ then go to step 4

 otherwise go to step 5

Step 4: if $a > c$ then

 print a is largest otherwise

 print c is largest

step 5: if $b > c$ then

 print b is largest otherwise

 print c is largest

Step 6: end

```
//To add two numbers
```

```
Algorithm add(a,b)
```

```
begin
```

```
read a,b,c;
```

```
c := a+b;
```

```
print c;
```

```
end
```

```
// To swap two numbers
```

```
Algorithm swap(a,b)
```

```
begin
```

```
temp <- a;
```

```
a <- b;
```

```
b <- temp;
```

```
end
```

Line Count and Operation Count Methods:

- In order to calculate time complexity we use frequency count or step count and operation count method.
- The number of times the basic operation will get executed or the frequency of the basic operation is Line count or Frequency count method.

Operation Count Method:

- Every algorithm will have some basic operation (maximum execution time) to solve a given problem.
- Finding the execution time of basic operation is Operation count.

Example:

```
int maximum( int a[ ] , int n)
```

```
{
```

```
max = a[0];
```

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

```
{
```

```
if(a[i] > max) // basic operation
```

```
max = a[i];
```

```
}
```

```
return max;
```

```
}
```

Basic operation count : $b = 1$ (Comparison operation)

Line count : $C(n) = n - 1$

Time Complexity : The product of basic operation and line count against n (size of the array)

$$T(n) = b \cdot C(n)$$
$$= 1 \cdot (n-1)$$
$$= n-1$$

Frequency Count Method:

- In order to calculate time complexity we use frequency count or step count method.
- It specifies the number of times a statement to be executed.

Rules to calculate frequency count or step count :

- For comments, declarations step count is 0.
- For return and assignment statement step count is 1.
Ex: $x = 10$
- Consider higher order exponent only
- Do not consider constants.

Example: $2x^4 + 4x^3 + 5x^2$

step count of above statement is 4 (considering higher order exponent). In written we will write $O(n^4)$.

Problem 1: Write an algorithm to calculate sum of elements in an array and calculate step count for the same.

Algorithm:

Algorithm Sum(A, n)

begin

```

s:=0 // Step count is 1
for( i=0;i<n;i++) // step count is n+1
begin
s = s + A[i]; // step count is n
end
return s; // step count is 1
end

```

Total frequency count is : $1+n+1+n+1$

$2n+3$

Ignore the constants and consider the higher order exponential

So 2 and 3 are ignored

Step count is : $O(n)$

Space Complexity:

Array contain N elements : n

N is 1

S is 1

I is 1

$S(n) = n+3$

$O(n)$

Problem 2: Write an algorithm for square matrix and calculate square matrix.

Algorithm Add(A, B, n)

Begin

```
for(i=0;i<n;i++) // n+1
```

begin

```
for(j=0;j<n;j++) // n*(n+1)
```

begin

```
C[i,j] = A[i,j] + B[i,j]; //n
```

end

end

end

Time complexity:

$$n+1+n*(n+1)+n = n+1+n^2+n+n = n^2+3n+1$$

$$O(n^2)$$

Space complexity:

$$A - n^2$$

$$B - n^2$$

$$C - n^2$$

$$N - 1$$

$$I - 1$$

$$J - 1$$

$$S(n) = 3n^2 + 3$$

$$O(n^2)$$

1.2 Analysis of Algorithm

Analysis of Algorithm:

The **analysis of algorithms** is the process of finding the computational complexity of algorithms—the amount of time, storage, or other resources needed to execute them.

- It involves studying their behaviour under different scenarios, typically categorized as best-case, average-case, and worst-case scenarios.

Best Case:

- This refers to the scenario in which the algorithm performs optimally, achieving the lowest possible time or space complexity.
- Best-case analysis often involves identifying inputs for which the algorithm runs most efficiently. However, the best-case scenario is not always the most common or representative of real-world usage.

Average Case:

- Average-case analysis considers the performance of an algorithm when given input data that is randomly distributed across all possible inputs.
- It involves calculating the expected time or space complexity by considering the probabilities of various input distributions. This analysis can provide a more realistic assessment of an algorithm's efficiency compared to the best-case scenario.

Worst Case:

- The worst-case scenario represents the input data that causes the algorithm to perform the least efficiently, resulting in the highest time or space complexity.

- It's essential to analyse worst-case performance because it guarantees an upper bound on the algorithm's performance regardless of the input.
- In many practical situations, worst-case analysis is crucial for ensuring that the algorithm doesn't degrade to unacceptable performance levels.

Key Points:

- When analysing algorithms, it's common to focus on worst-case performance because it provides a guarantee on how the algorithm will behave in the most challenging situations.
- However, average-case analysis is also important for understanding real-world performance, especially when the input data is not uniformly distributed or when the worst-case scenario is rare.
- Different algorithms may have different best, average, and worst-case complexities, which can significantly impact their suitability for specific tasks or datasets. It's essential to consider these factors when selecting an algorithm for a particular problem.

Example 1:

Linear Search:

Linear Search is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set.

Algorithm:

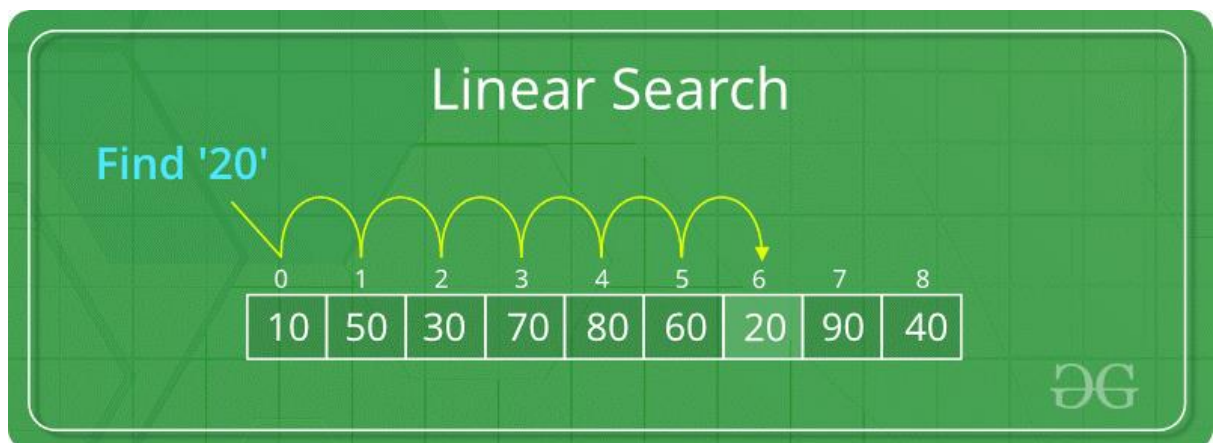
Algorithm linearSearch(int array[], int size, int target)

begin

 for (int i = 0; i < size; i++)

 begin


```
    if (array[i] == target)
begin
    return i; // Return index if target found
end
end
return -1; // Return -1 if target not found
end
```



Analysis of Linear Search Algorithm:

Time Complexity:

Best Case: In the best case, the key might be present at the first index. So the best case complexity is $O(1)$

Worst Case: In the worst case, the key might be present at the last index i.e., opposite to the end from which the search has started in the list. So the worst-case complexity is $O(N)$ where N is the size of the list.

Average Case: $O(N)$

Auxiliary Space: $O(1)$ as except for the variable to iterate through the list, no other variable is used.

Advantages of Linear Search:

Linear search can be used irrespective of whether the array is sorted or not.

It can be used on arrays of any data type.

Does not require any additional memory.

It is a well-suited algorithm for small datasets.

Drawbacks of Linear Search:

Linear search has a time complexity of $O(N)$, which in turn makes it slow for large datasets.

Not suitable for large arrays.

Example 2:

Binary Search:

Binary Search is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half and the correct interval to find is decided based on the searched value and the mid value of the interval.



Properties of Binary Search:

- Binary search is performed on the sorted data structure for example sorted array.
- Searching is done by dividing the array into two halves.
- It utilizes the divide-and-conquer approach to find an element.

Applications of Binary Search:

- The binary search operation is applied to any sorted array for finding any element.
- Binary search is more efficient and faster than linear search.
- In real life, binary search can be applied in the dictionary.
- Binary search is also used to debug a linear piece of code.
- Binary search is also used to find if a number is a square of another or not.

Time Complexity of Binary Search Algorithm:

Best Case Time Complexity of Binary Search Algorithm: $O(1)$

Best case is when the element is at the middle index of the array. It takes only one comparison to find the target element. So the best case complexity is $O(1)$.

Average Case Time Complexity of Binary Search Algorithm: $O(\log N)$

Case1: Element is present in the array

Case2: Element is not present in the array.

There are N Case1 and 1 Case2. So total number of cases = $N+1$. Now notice the following:

An element at index $N/2$ can be found in 1 comparison

Elements at index $N/4$ and $3N/4$ can be found in 2 comparisons.

Elements at indices $N/8$, $3N/8$, $5N/8$ and $7N/8$ can be found in 3 comparisons and so on.

Based on this we can conclude that elements that require:

1 comparison = 1

2 comparisons = 2

3 comparisons = 4

x comparisons = 2^{x-1} where x belongs to the range $[1, \log N]$ because maximum comparisons = maximum time N can be halved = maximum comparisons to reach 1st element = $\log N$.

So, total comparisons

= $1 \cdot (\text{elements requiring 1 comparisons}) + 2 \cdot (\text{elements requiring 2 comparisons}) + \dots + \log N \cdot (\text{elements requiring } \log N \text{ comparisons})$

= $1 \cdot 1 + 2 \cdot 2 + 3 \cdot 4 + \dots + \log N \cdot (2^{\log N - 1})$

= $2 \log N \cdot (\log N - 1) + 1$

= $N \cdot (\log N - 1) + 1$

Total number of cases = $N+1$.

Therefore, the average complexity = $(N \cdot (\log N - 1) + 1) / (N+1) = N \cdot \log N / (N+1) + 1 / (N+1)$. Here the dominant term is $N \cdot \log N / (N+1)$ which is approximately $\log N$.

So the average case complexity is $O(\log N)$.

Worst Case Time Complexity of Binary Search Algorithm: $O(\log N)$

The worst case will be when the element is present in the first position. As seen in the average case, the comparison required to reach the first element is $\log N$. So the time complexity for the worst case is $O(\log N)$.

Algorithm:

Algorithm BinarySearch {

```
// Returns index of x if it is present in arr[].
int binarySearch(int arr[], int x)
{
    int l = 0, r = arr.length - 1;
    while (l <= r) {
        int m = l + (r - l) / 2;

        // Check if x is present at mid
        if (arr[m] == x)
            return m;

        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // If we reach here, then element was
    // not present
    return -1;
}
```

1.3 Asymptotic Notations (O, Omega, Theta)

The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measured with the help of asymptotic notations.

An algorithm may not have the same performance for different types of inputs. With the increase in the input size, the performance will change.

The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.

Asymptotic Notations

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

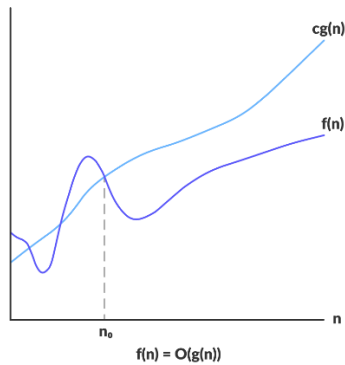
When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations:

- **Big-O notation**
- **Omega notation**
- **Theta notation**

Big-O Notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.

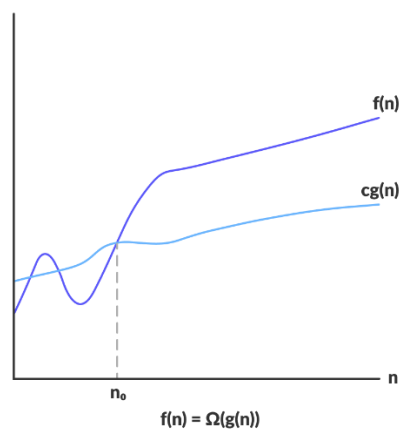


$$O(g(n)) = \{ f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}$$

Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.

Omega Notation (Ω -notation)

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

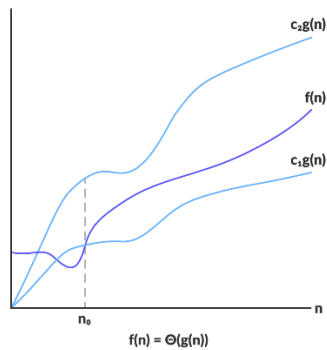


$$\Omega(g(n)) = \{ f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \}$$

For any value of n , the minimum time required by the algorithm is given by Omega $\Omega(g(n))$.

Theta Notation (Θ -notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.



For a function $g(n)$, $\Theta(g(n))$ is given by the relation:

$$\Theta(g(n)) = \{ f(n): \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0 \}$$

If a function $f(n)$ lies anywhere in between $c_1g(n)$ and $c_2g(n)$ for all $n \geq n_0$, then $f(n)$ is said to be asymptotically tight bound.

1.4 Recursive Algorithms

Recursion is the action of a function calling itself either directly or indirectly, and the associated function is known as a recursive function

A recursive algorithm calls itself with smaller input values and, after performing basic operations on the returned value for the minor input, provides the result for the current input. A recursive method can solve a problem if smaller versions of the same problem can be solved by applying solutions to them, and the smaller versions decrease to easily solvable examples.

Characteristics of Recursion

Recursion's characteristics include:

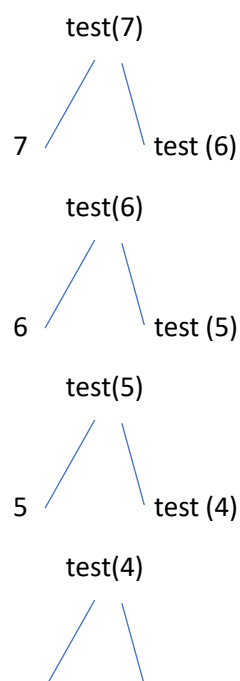
Repeating the same actions with different inputs.

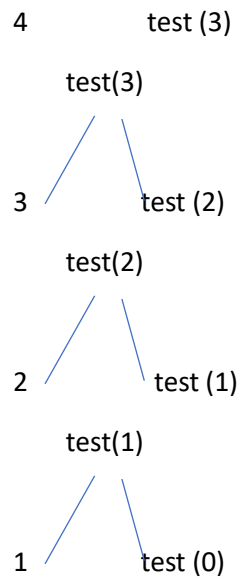
To make the issue smaller at each phase, we experiment with smaller inputs.

A base condition must stop the recursion; otherwise, an infinite loop would result.

```
int test(int n)
{
    if (n > 0) // base case
    {
        printf("%d",n);
        test(n-1);
    }
}
```

Recursive Tree:





Execution Time

```

int test(int n)
{
    if (n > 0) // base case ----- 1
    {
        printf("%d",n); ----- 1
        test(n-1); ----- n+1
    }
}

```

Printf statement will get executed for 1 time

Test(n) will get executed recursively by n+1 times

$$\begin{aligned}
 f(n) &= n+1 \\
 &= n^1 + n^0 \\
 &= n \text{ (consider higher order exponential)}
 \end{aligned}$$

$$f(n) = O(n)$$

1.5 Analysis using recurrence relations

Recursion is the action of a function calling itself either directly or indirectly, and the associated function is known as a recursive function

A recursive algorithm calls itself with smaller input values and, after performing basic operations on the returned value for the minor input, provides the result for the current input. A recursive method can solve a problem if smaller versions of the same problem can be solved by applying solutions to them, and the smaller versions decrease to easily solvable examples.

Characteristics of Recursion

Recursion's characteristics include:

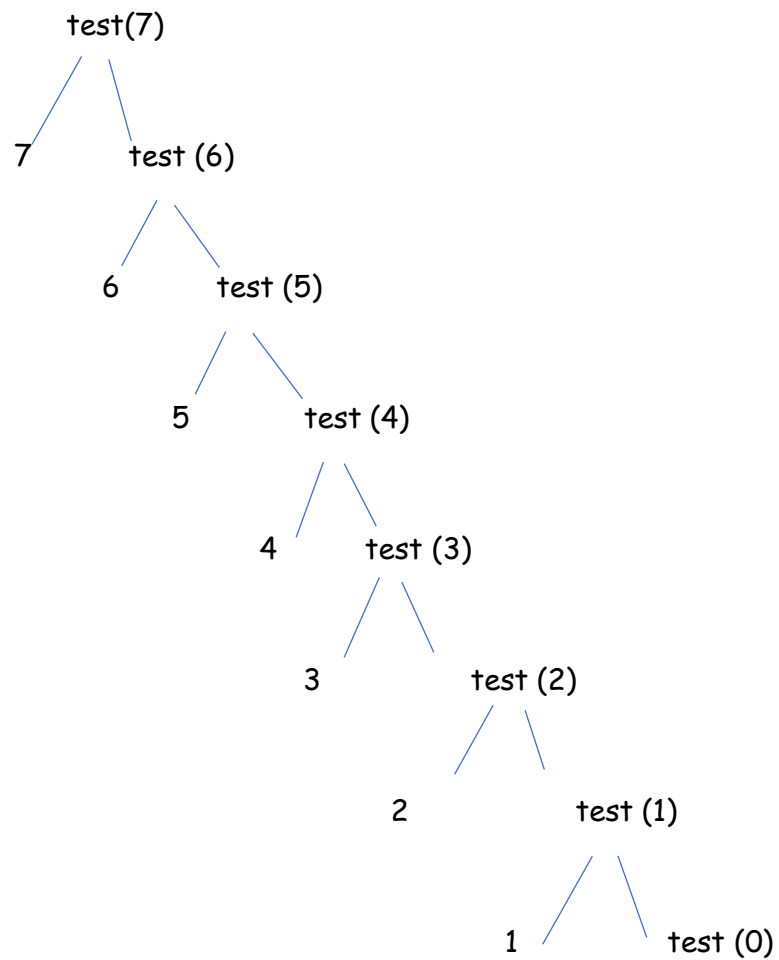
Repeating the same actions with different inputs.

To make the issue smaller at each phase, we experiment with smaller inputs.

A base condition must stop the recursion; otherwise, an infinite loop would result.

```
int test(int n)
{
    if (n > 0) // base case
    {
        printf("%d",n);
        test(n-1);
    }
}
```

Recursive Tree:



Execution Time

```
int test(int n)
{
    if (n > 0) // base case ----- 1
    {
        printf("%d",n); ----- 1
        test(n-1); ----- n+1
    }
}
```

Printf statement will get executed for 1 time

Test(n) will get executed recursively by n+1 times

$$f(n) = n+1$$

$$= n^1 + n^0$$

$$= n \text{ (consider higher order exponential)}$$

$$f(n) = O(n)$$

- A recurrence relation is a mathematical expression that defines a sequence in terms of its previous terms. In the context of algorithmic analysis, it is often used to model the time complexity of recursive algorithms.
- General form of a Recurrence Relation: $a_{\{n\}} = f(a_{\{n-1\}}, a_{\{n-2\}}, \dots, a_{\{n-k\}})$ where f is a function that defines the relationship between the current term and the previous terms.

Following are some of the examples of recurrence relations based on linear recurrence relation.

$$T(n) = T(n-1) + n \text{ for } n > 0 \text{ and } T(0) = 1$$

These types of recurrence relations can be easily solved using substitution method.

For example,

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

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

$$= T(n-k) + (n-(k-1)) \dots (n-1) + n$$

Substituting $k = n$, we get

$$T(n) = T(0) + 1 + 2 + \dots + n = n(n+1)/2 = O(n^2)$$

Master theorem

In the analysis of algorithms, the **master theorem** provides a cookbook solution in asymptotic terms (using Big O notation) for recurrence relations of types that occur in the analysis of many divide and conquer algorithms. It was popularized by the canonical algorithms textbook *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein, which introduces and proves it in sections 4.3 and 4.4, respectively. Nevertheless, not all recurrence relations can be solved with the use of the master theorem; its generalizations include the Akra–Bazzi method.

Introduction

Consider a problem that can be solved using a recursive algorithm such as the following:

```

procedure T( n : size of problem ) defined as:
    if n < 1 then exit

    Do work of amount f(n)

    T(n/b)
    T(n/b)
    ...repeat for a total of a times...
    T(n/b)
end procedure

```

In the above algorithm we are dividing the problem into a number of sub problems recursively, each sub problem being of size n/b . This can be visualized as building a call tree with each node of the tree as an instance of one recursive call and its child nodes being instances of subsequent calls. In the above example, each node would have a number of child nodes. Each node does an amount of work that corresponds to the size of the sub problem n passed to that instance of the recursive call and given by $f(n)$. For example, if each recursive call is doing a comparison sort, then the amount of work done by each node in the tree is at least $O(n \log n)$. The total amount of work done by the entire tree is the sum of the work performed by all the nodes in the tree.

Algorithm such as above can be represented as recurrence relationship $T(n) = a T\left(\frac{n}{b}\right) + f(n)$. This recursive relationship can be successively substituted in to itself and expanded to obtain expression for total amount of work done.^[1]

The original Master theorem allows to easily calculate run time of such a recursive algorithm in Big O notation without doing expansion of above recursive relationship. A generalized form of Master Theorem by Akra and Bazzi introduced in 1998 is applicable on wide number of cases that occur in practice.

Generic form

The master theorem concerns recurrence relations of the form:

$$T(n) = a T\left(\frac{n}{b}\right) + f(n) \quad \text{where } a \geq 1, b > 1$$

In the application to the analysis of a recursive algorithm, the constants and function take on the following significance:

- n is the size of the problem.
- a is the number of subproblems in the recursion.
- n/b is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)
- $f(n)$ is the cost of the work done outside the recursive calls, which includes the cost of dividing the problem and the cost of merging the solutions to the subproblems.

It is possible to determine an asymptotic tight bound in these three cases:

Case 1

Generic form

If $f(n) = O(n^{\log_b(a)-\epsilon})$ for some constant $\epsilon > 0$ (using Big O notation) it follows that:

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

Example

$$T(n) = 8T\left(\frac{n}{2}\right) + 1000n^2$$

As one can see in the formula above, the variables get the following values:

$$a = 8, b = 2, f(n) = 1000n^2, \log_b a = \log_2 8 = 3$$

Now we have to check that the following equation holds:

$$f(n) = O(n^{\log_b a - \epsilon})$$

$$1000n^2 = O(n^{3-\epsilon})$$

If we choose $\epsilon = 1$, we get:

$$1000n^2 = O(n^{3-1}) = O(n^2)$$

Since this equation holds, the first case of the master theorem applies to the given recurrence relation, thus resulting in the conclusion:

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

If we insert the values from above, we finally get:

$$T(n) = \Theta(n^3)$$

Thus the given recurrence relation $T(n)$ was in $\Theta(n^3)$.

(This result is confirmed by the exact solution of the recurrence relation, which is $T(n) = 1001n^3 - 1000n^2$, assuming $T(1) = 1$).

Case 2

Generic form

If it is true, for some constant $k \geq 0$, that:

$$f(n) = \Theta \left(n^{\log_b a} \log^k n \right)$$

it follows that:

$$T(n) = \Theta \left(n^{\log_b a} \log^{k+1} n \right)$$

Example

$$T(n) = 2T\left(\frac{n}{2}\right) + 10n$$

As we can see in the formula above the variables get the following values:

$$a = 2, b = 2, k = 0, f(n) = 10n, \log_b a = \log_2 2 = 1$$

Now we have to check that the following equation holds (in this case $k=0$):

$$f(n) = \Theta \left(n^{\log_b a} \right)$$

If we insert the values from above, we get:

$$10n = \Theta \left(n^1 \right) = \Theta(n)$$

Since this equation holds, the second case of the master theorem applies to the given recurrence relation, thus resulting in the conclusion:

$$T(n) = \Theta \left(n^{\log_b a} \log^{k+1} n \right)$$

If we insert the values from above, we finally get:

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

Thus the given recurrence relation $T(n)$ was in $\Theta(n \log n)$.

(This result is confirmed by the exact solution of the recurrence relation, which is $T(n) = n + 10n \log_2 n$, assuming $T(1) = 1$.)

Case 3

Generic form

If it is true that:

$$f(n) = \Omega \left(n^{\log_b(a)+\epsilon} \right) \text{ for some constant } \epsilon > 0$$

and if it is also true that:

$$af\left(\frac{n}{b}\right) \leq cf(n) \text{ for some constant } c < 1 \text{ and sufficiently large } n$$

it follows that:

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

Example

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

As we can see in the formula above the variables get the following values:

$$a = 2, b = 2, f(n) = n^2, \log_b a = \log_2 2 = 1$$

Now we have to check that the following equation holds:

$$f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$$

If we insert the values from above, and choose $\epsilon = 1$, we get:

$$n^2 = \Omega\left(n^{1+1}\right) = \Omega\left(n^2\right)$$

Since this equation holds, we have to check the second condition, namely if it is true that:

$$af\left(\frac{n}{b}\right) \leq cf(n)$$

If we insert once more the values from above, we get the number :

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

If we choose $c = \frac{1}{2}$, it is true that:

$$\frac{1}{2}n^2 \leq \frac{1}{2}n^2 \quad \forall n \geq 1$$

So it follows:

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

If we insert once more the necessary values, we get:

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

Thus the given recurrence relation $T(n)$ was in $\Theta(n^2)$, that complies with the $f(n)$ of the original formula.

(This result is confirmed by the exact solution of the recurrence relation, which is $T(n) = 2n^2 - n$, assuming $T(1) = 1$.)

Inadmissible equations

The following equations cannot be solved using the master theorem.^[2]

- $T(n) = 2^n T\left(\frac{n}{2}\right) + n^n$

a is not a constant

- $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n}$

non-polynomial difference between $f(n)$ and $n^{\log_b a}$ (See Below)

- $T(n) = 0.5T\left(\frac{n}{2}\right) + n$

$a < 1$ cannot have less than one sub problem

- $T(n) = 64T\left(\frac{n}{8}\right) - n^2 \log n$

$f(n)$ is not positive

- $T(n) = T\left(\frac{n}{2}\right) + n(2 - \cos n)$

case 3 but regularity violation.

In the second inadmissible example above, the difference between $f(n)$ and $n^{\log_b a}$ can be expressed with the ratio $\frac{f(n)}{n^{\log_b a}} = \frac{\frac{n}{\log n}}{n^{\log_2 2}} = \frac{n}{n \log n} = \frac{1}{\log n}$. It is clear that $\frac{1}{\log n} < n^\epsilon$ for any constant $\epsilon > 0$. Therefore, the difference is not polynomial and the Master Theorem does not apply.

Application to common algorithms

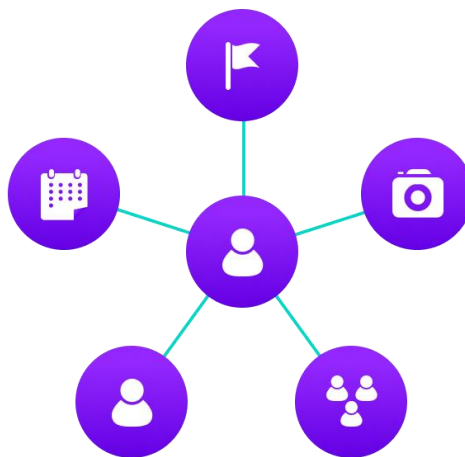
Algorithm	Recurrence Relationship	Run time	Comment
Binary search	$T(n) = T\left(\frac{n}{2}\right) + O(1)$	$O(\log(n))$	Apply Master theorem where $f(n) = n^c$ ^[3]
Binary tree traversal	$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$	$O(n)$	Apply Master theorem where $f(n) = n^c$ ^[3]
Optimal Sorted Matrix Search	$T(n) = 2T\left(\frac{n}{2}\right) + O(\log(n))$	$O(n)$	Apply Akra-Bazzi theorem for and to get $\Theta(2n - \log(n))$
Merge Sort	$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$	$O(n \log(n))$	

1.9 Graphs

A graph data structure is a collection of nodes that have data and are connected to other nodes.

Let's try to understand this through an example. On facebook, everything is a node. That includes User, Photo, Album, Event, Group, Page, Comment, Story, Video, Link, Note...anything that has data is a node.

Every relationship is an edge from one node to another. Whether you post a photo, join a group, like a page, etc., a new edge is created for that relationship.



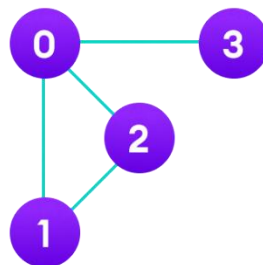
Example of graph data structure

All of facebook is then a collection of these nodes and edges. This is because facebook uses a graph data structure to store its data.

More precisely, a graph is a data structure (V, E) that consists of

A collection of vertices V

A collection of edges E , represented as ordered pairs of vertices (u,v)



Vertices and edges

In the graph,

$$V = \{0, 1, 2, 3\}$$

$$E = \{(0,1), (0,2), (0,3), (1,2)\}$$

$$G = \{V, E\}$$

Graph Terminology

Adjacency: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them.

Path: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.

Directed Graph: A graph in which an edge (u,v) doesn't necessarily mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge.

Graph Representation

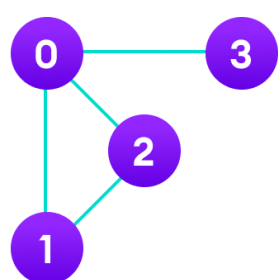
Graphs are commonly represented in two ways:

1. Adjacency Matrix

An adjacency matrix is a 2D array of $V \times V$ vertices. Each row and column represent a vertex.

If the value of any element $a[i][j]$ is 1, it represents that there is an edge connecting vertex i and vertex j .

The adjacency matrix for the graph we created above is



	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

Graph adjacency matrix

Since it is an undirected graph, for edge (0,2), we also need to mark edge (2,0); making the adjacency matrix symmetric about the diagonal.

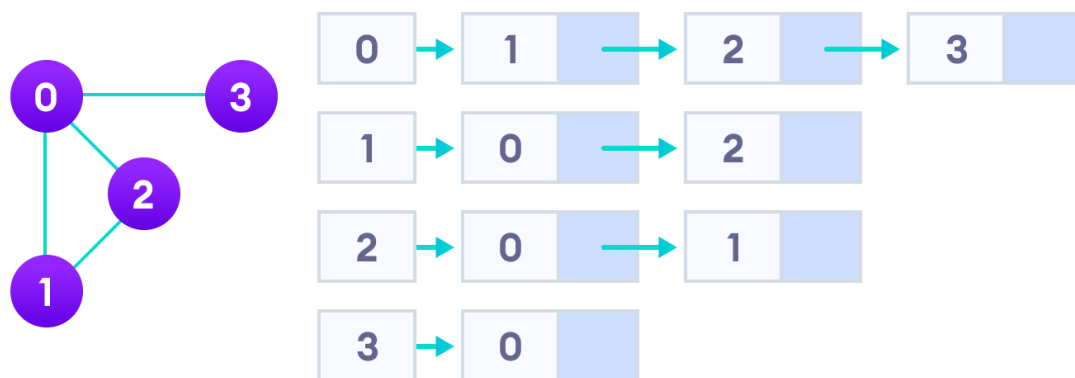
Edge lookup(checking if an edge exists between vertex A and vertex B) is extremely fast in adjacency matrix representation but we have to reserve space for every possible link between all vertices($V \times V$), so it requires more space.

2. Adjacency List

An adjacency list represents a graph as an array of linked lists.

The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

The adjacency list for the graph we made in the first example is as follows:



Adjacency list representation

An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a graph with millions of vertices, this can mean a lot of saved space.

Graph Operations

The most common graph operations are:

Check if the element is present in the graph

Graph Traversal

Add elements(vertex, edges) to graph

Finding the path from one vertex to another

Graphs (BFS ,DFS)

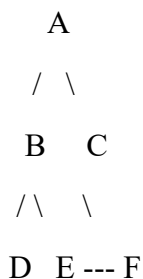
In Java, you can implement various graph traversal techniques such as Depth-First Search (DFS) and Breadth-First Search (BFS) using different data structures like adjacency lists or adjacency matrices. Here, I'll provide examples for both DFS and BFS using adjacency lists in Java.

1. Depth-First Search (DFS):

DFS explores as far as possible along each branch before backtracking.

It's often used for topological sorting, cycle detection, and solving maze problems.

Example graph:



DFS traversal starting from node 'A':

A, B, D, E, F, C

DFS explores as far as possible along each branch before backtracking. Here's how you can implement DFS in Java using recursion:

```
import java.util.*;
```

```
public class DFSGraphTraversal {  
    private Map<Integer, List<Integer>> graph;  
  
    public DFSGraphTraversal(Map<Integer, List<Integer>> graph) {  
        this.graph = graph;  
    }  
  
    public void dfs(int start) {  
        Set<Integer> visited = new HashSet<>();  
        dfsHelper(start, visited);  
    }  
}
```

```

private void dfsHelper(int node, Set<Integer> visited) {
    visited.add(node);
    System.out.print(node + " ");

    List<Integer> neighbors = graph.getOrDefault(node, new ArrayList<>());
    for (int neighbor : neighbors) {
        if (!visited.contains(neighbor)) {
            dfsHelper(neighbor, visited);
        }
    }
}

public static void main(String[] args) {
    Map<Integer, List<Integer>> graph = new HashMap<>();
    graph.put(0, Arrays.asList(1, 2));
    graph.put(1, Arrays.asList(2));
    graph.put(2, Arrays.asList(0, 3));
    graph.put(3, Arrays.asList(3));

    DFSGraphTraversal dfsTraversal = new DFSGraphTraversal(graph);
    System.out.println("DFS Traversal:");
    dfsTraversal.dfs(2);
}
}

```

2. Breadth-First Search (BFS):

Graph traversal techniques are algorithms used to visit and explore all the nodes in a graph. Two common graph traversal techniques are Breadth-First Search (BFS) and Depth-First Search (DFS). Let's discuss these techniques with examples of graphs:

Breadth-First Search (BFS):

A

```
    /  \
   B    C
  / \   \
 D  E --- F
```

BFS traversal starting from node 'A':

Level 0: A

Level 1: B, C

Level 2: D, E, F

BFS explores nodes level by level, visiting all the neighbors of a node before moving on to the next level.

It's often used to find the shortest path in unweighted graphs.

Example graph:

BFS explores nodes level by level, visiting all the neighbors of a node before moving on to the next level. Here's how you can implement BFS in Java using a queue:

```
import java.util.*;
```

```
public class BFSGraphTraversal {

    private Map<Integer, List<Integer>>> graph;

    public BFSGraphTraversal(Map<Integer, List<Integer>>> graph) {
        this.graph = graph;
    }

    public void bfs(int start) {
        Set<Integer> visited = new HashSet<>();
        Queue<Integer> queue = new LinkedList<>();
        queue.offer(start);
        visited.add(start);

        while (!queue.isEmpty()) {
            int node = queue.poll();
```



```
System.out.print(node + " ");
```

```
List<Integer> neighbors = graph.getDefault(node, new ArrayList<>());
```

```
for (int neighbor : neighbors) {
```

```
    if (!visited.contains(neighbor)) {
```

```
        queue.offer(neighbor);
```

```
        visited.add(neighbor);
```

```
    }
```

```
}
```

```
}
```

```
}
```

```
public static void main(String[] args) {
```

```
    Map<Integer, List<Integer>> graph = new HashMap<>();
```

```
    graph.put(0, Arrays.asList(1, 2));
```

```
    graph.put(1, Arrays.asList(2));
```

```
    graph.put(2, Arrays.asList(0, 3));
```

```
    graph.put(3, Arrays.asList(3));
```

```
    BFSGraphTraversal bfsTraversal = new BFSGraphTraversal(graph);
```

```
    System.out.println("BFS Traversal:");
```

```
    bfsTraversal.bfs(2);
```

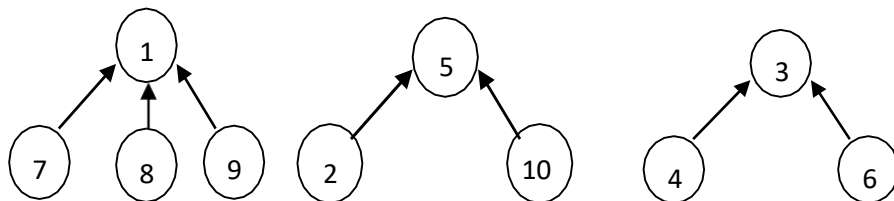
```
}
```

```
}
```

Set Representation:

Sets and Disjoint Set Union:

Disjoint Set Union: Considering a set $S = \{1, 2, 3, \dots, 10\}$ (when $n=10$), then elements can be partitioned into three disjoint sets $s_1 = \{1, 7, 8, 9\}$, $s_2 = \{2, 5, 10\}$ and $s_3 = \{3, 4, 6\}$. Possible tree representations are:



In this representation each set is represented as a tree. Nodes are linked from the child to parent rather than usual method of linking from parent to child.

The operations on these sets are:

1. Disjoint setunion
2. Find(i)
3. MinOperation
4. Delete
5. Intersect

1. Disjoint Set union:

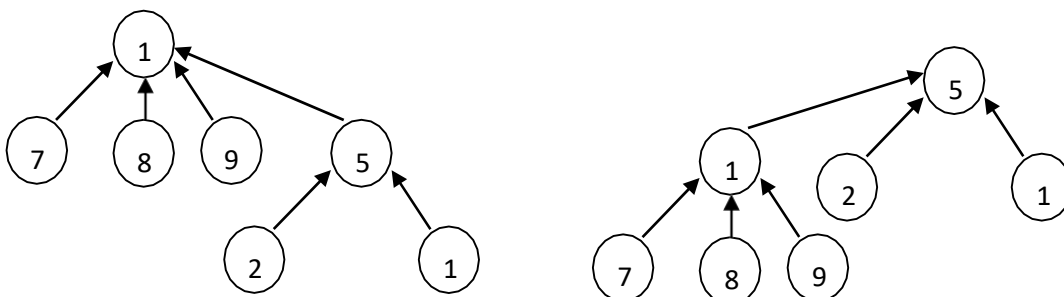
If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j =$ all the elements x such that x is in S_i or S_j . Thus $S_1 \cup S_2 = \{1, 7, 8, 9, 2, 5, 10\}$.

2. Find(i):

Given the element I , find the set containing i . Thus, 4 is in set S_3 , 9 is in S_1 .

UNION operation:

Union(i, j) requires two tree with roots i and j be joined. $S_1 \cup S_2$ is obtained by making any one of the sets as sub tree of other.



Simple Algorithm for Union:

Algorithm Union(i,j)

{

//replace the disjoint sets with roots i and j, I not equal to j by their union Integer i,j;

P[j] :=i;

}

Example:

Implement following sequence of operations Union(1,3),Union(2,5),Union(1,2)

Solution:

Initially parent array contains zeros.

0	0	0	0	0	0
1	2	3	4	5	6

1. After performing union(1,3)operationParent[3]:=1

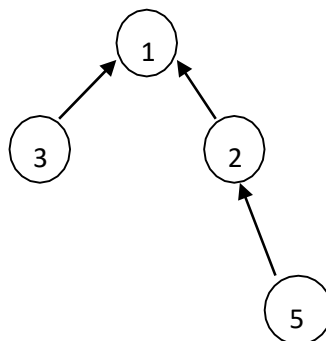
0	0	1	0	0	0
1	2	3	4	5	6

2. After performing union(2,5)operationParent[5]:=2

0	0	1	0	2	0
1	2	3	4	5	6

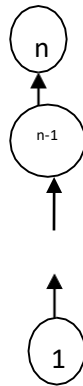
3. After performing union(1,2)operationParent[2]:=1

0	1	1	0	2	0
1	2	3	4	5	6



Process the following sequence of union operations $\text{Union}(1,2), \text{Union}(2,3), \dots, \text{Union}(n-1,n)$

Degenerate Tree:



The time taken for $n-1$ unions is $O(n)$.

Find(i) operation: determines the root of the tree containing element i . Simple Algorithm for Find:

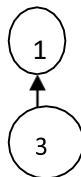
Algorithm Find(i)

```

{
    j:=i;
    while(p[j]>0) do
        j:=p[j]; return j;
}
  
```

Find Operation: Find(i) implies that it finds the root node of i^{th} node, in other words it returns the name of the set i .

Example: Consider the Union(1,3)

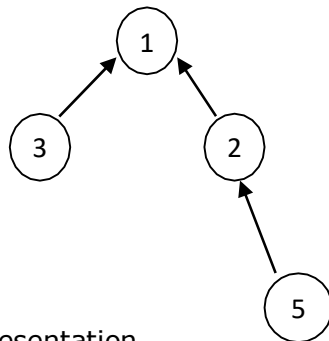


Find(1)=0

Find(3)=1, since its parent is 1. (i.e, root is 1)

Example:

Considering



Array Representation

P[i]	0	1	1	2
i	1	2	3	5

Find(5)=1

Find(2)=1

Find(3)=1

The root node represents all the nodes in the tree. Time Complexity of „n“ find operations is $O(n^2)$.

To improve the performance of union and find algorithms by avoiding the creation of degenerate tree. To accomplish this, we use weighting rule for Union(i,j).

Weighting Rule for Union(i,j)

```
1  Algorithm WeightedUnion(i, j)
2  // Union sets with roots i and j,  $i \neq j$ , using the
3  // weighting rule.  $p[i] = -count[i]$  and  $p[j] = -count[j]$ .
4  {
5      temp := p[i] + p[j];
6      if (p[i] > p[j]) then
7          { // i has fewer nodes.
8              p[i] := j; p[j] := temp;
9          }
10     else
11         { // j has fewer or equal nodes.
12             p[j] := i; p[i] := temp;
13         }
14 }
```

Union algorithm with weighting rule

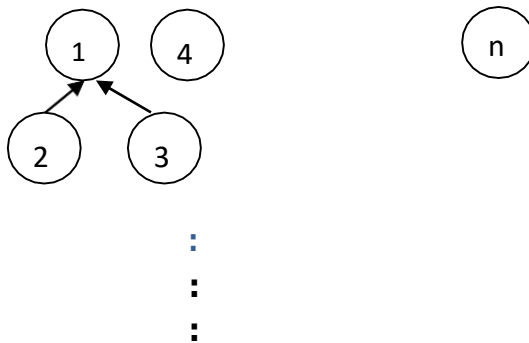
Tree obtained with
weighted Initially



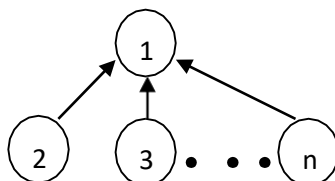
Union(1,2)



Union(1,3)



Union(1,n)



Collapsing Rule for Find(i)

```

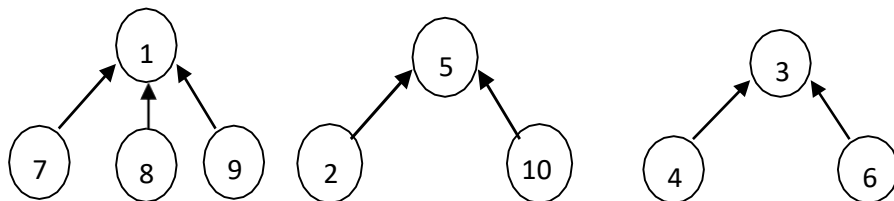
1  Algorithm CollapsingFind(i)
2  // Find the root of the tree containing element i. Use the
3  // collapsing rule to collapse all nodes from i to the root.
4  {
5      r := i;
6      while (p[r] > 0) do r := p[r]; // Find the root.
7      while (i ≠ r) do // Collapse nodes from i to root r.
8      {
9          s := p[i]; p[i] := r; i := s;
10     }
11     return r;
12 }
```

Find algorithm with collapsing rule

Set Representation:

Sets and Disjoint Set Union:

Disjoint Set Union: Considering a set $S = \{1, 2, 3, \dots, 10\}$ (when $n=10$), then elements can be partitioned into three disjoint sets $s_1 = \{1, 7, 8, 9\}$, $s_2 = \{2, 5, 10\}$ and $s_3 = \{3, 4, 6\}$. Possible tree representations are:



In this representation each set is represented as a tree. Nodes are linked from the child to parent rather than usual method of linking from parent to child.

The operations on these sets are:

1. Disjoint setunion
2. Find(i)
3. MinOperation
4. Delete
5. Intersect

1. Disjoint Set union:

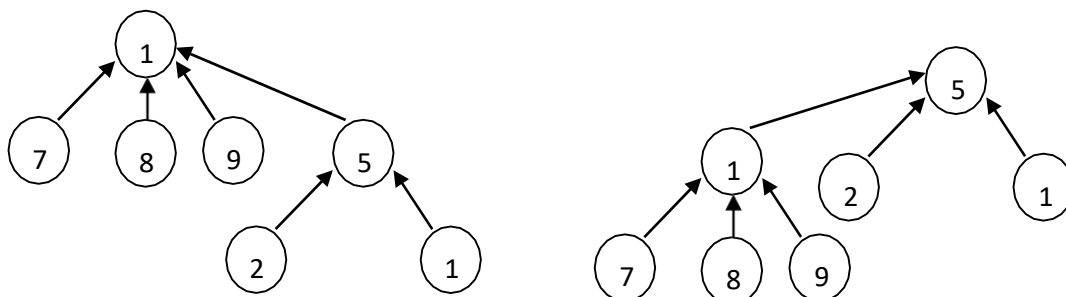
If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j =$ all the elements x such that x is in S_i or S_j . Thus $S_1 \cup S_2 = \{1, 7, 8, 9, 2, 5, 10\}$.

2. Find(i):

Given the element I , find the set containing i . Thus, 4 is in set S_3 , 9 is in S_1 .

UNION operation:

Union(i, j) requires two tree with roots i and j be joined. $S_1 \cup S_2$ is obtained by making any one of the sets as sub tree of other.



Simple Algorithm for Union:

Algorithm Union(i,j)

```
{  
  //replace the disjoint sets with roots i and j, I not equal to j by their  
  union Integer i,j;  
  P[j] :=i;  
}
```

Example:

Implement following sequence of operations Union(1,3),Union(2,5),Union(1,2)

Solution:

Initially parent array contains zeros.

0	0	0	0	0	0
1	2	3	4	5	6

1. After performing union(1,3)operationParent[3]:=1

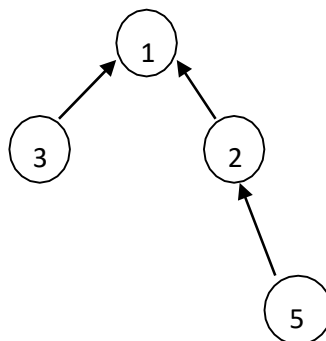
0	0	1	0	0	0
1	2	3	4	5	6

2. After performing union(2,5)operationParent[5]:=2

0	0	1	0	2	0
1	2	3	4	5	6

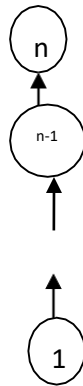
3. After performing union(1,2)operationParent[2]:=1

0	1	1	0	2	0
1	2	3	4	5	6



Process the following sequence of union operations $\text{Union}(1,2), \text{Union}(2,3), \dots, \text{Union}(n-1,n)$

Degenerate Tree:



The time taken for $n-1$ unions is $O(n)$.

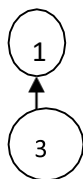
Find(i) operation: determines the root of the tree containing element i . Simple Algorithm for Find:

Algorithm Find(i)

```
{
    j:=i;
    while(p[j]>0) do
        j:=p[j]; return j;
}
```

Find Operation: Find(i) implies that it finds the root node of i^{th} node, in other words it returns the name of the set i .

Example: Consider the Union(1,3)

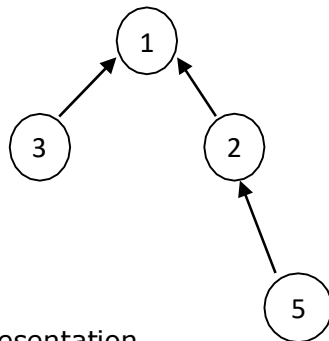


Find(1)=0

Find(3)=1, since its parent is 1. (i.e, root is 1)

Example:

Considering



Array Representation

P[i]	0	1	1	2
i	1	2	3	5

Find(5)=1

Find(2)=1

Find(3)=1

The root node represents all the nodes in the tree. Time Complexity of „n“ find operations is $O(n^2)$.

To improve the performance of union and find algorithms by avoiding the creation of degenerate tree. To accomplish this, we use weighting rule for Union(i,j).

Weighting Rule for Union(i,j)

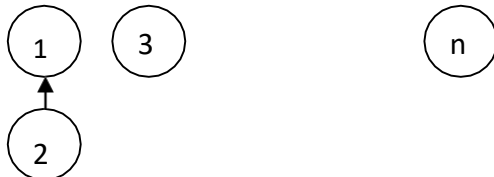
```
1  Algorithm WeightedUnion(i, j)
2  // Union sets with roots i and j, i ≠ j, using the
3  // weighting rule. p[i] = -count[i] and p[j] = -count[j].
4  {
5      temp := p[i] + p[j];
6      if (p[i] > p[j]) then
7          { // i has fewer nodes.
8              p[i] := j; p[j] := temp;
9          }
10     else
11         { // j has fewer or equal nodes.
12             p[j] := i; p[i] := temp;
13         }
14 }
```

Union algorithm with weighting rule

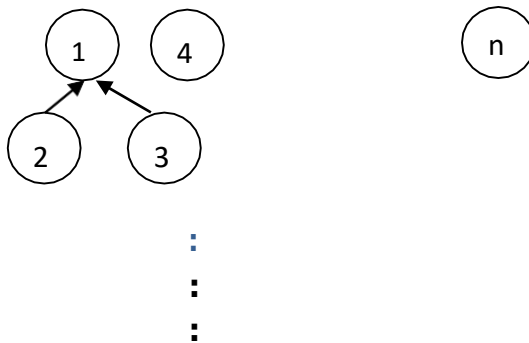
Tree obtained with
weighted Initially



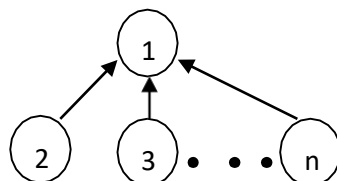
Union(1,2)



Union(1,3)



Union(1,n)



Collapsing Rule for Find(i)

```

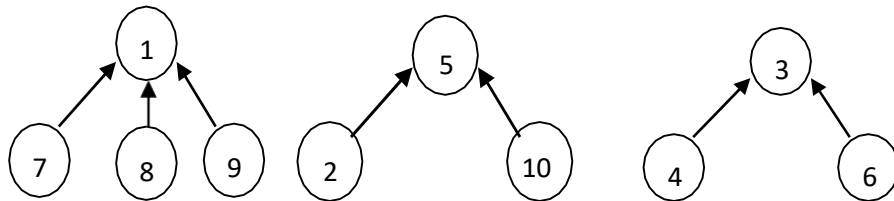
1  Algorithm CollapsingFind(i)
2  // Find the root of the tree containing element i. Use the
3  // collapsing rule to collapse all nodes from i to the root.
4  {
5      r := i;
6      while (p[r] > 0) do r := p[r]; // Find the root.
7      while (i ≠ r) do // Collapse nodes from i to root r.
8      {
9          s := p[i]; p[i] := r; i := s;
10     }
11     return r;
12 }
```

Find algorithm with collapsing rule

Set Representation:

Sets and Disjoint Set Union:

Disjoint Set Union: Considering a set $S = \{1, 2, 3, \dots, 10\}$ (when $n=10$), then elements can be partitioned into three disjoint sets $s_1 = \{1, 7, 8, 9\}$, $s_2 = \{2, 5, 10\}$ and $s_3 = \{3, 4, 6\}$. Possible tree representations are:



In this representation each set is represented as a tree. Nodes are linked from the child to parent rather than usual method of linking from parent to child.

The operations on these sets are:

1. Disjoint setunion
2. Find(i)
3. MinOperation
4. Delete
5. Intersect

1. Disjoint Set union:

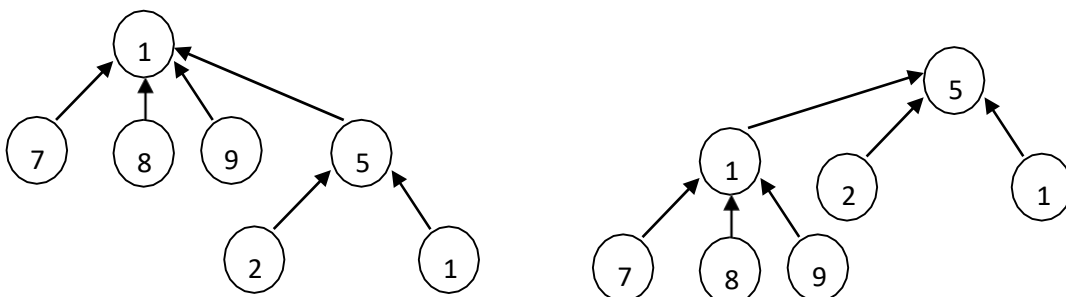
If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j =$ all the elements x such that x is in S_i or S_j . Thus $S_1 \cup S_2 = \{1, 7, 8, 9, 2, 5, 10\}$.

2. Find(i):

Given the element I , find the set containing i . Thus, 4 is in set S_3 , 9 is in S_1 .

UNION operation:

Union(i, j) requires two tree with roots i and j be joined. $S_1 \cup S_2$ is obtained by making any one of the sets as sub tree of other.



Simple Algorithm for Union:

Algorithm Union(i,j)

```
{  
  //replace the disjoint sets with roots i and j, I not equal to j by their  
  union Integer i,j;  
  P[j] :=i;  
}
```

Example:

Implement following sequence of operations Union(1,3),Union(2,5),Union(1,2)

Solution:

Initially parent array contains zeros.

0	0	0	0	0	0
1	2	3	4	5	6

1. After performing union(1,3)operationParent[3]:=1

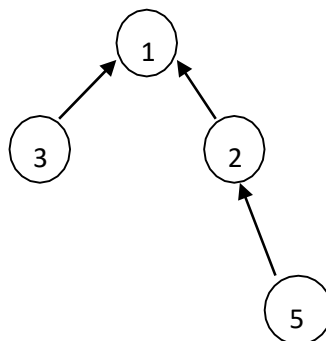
0	0	1	0	0	0
1	2	3	4	5	6

2. After performing union(2,5)operationParent[5]:=2

0	0	1	0	2	0
1	2	3	4	5	6

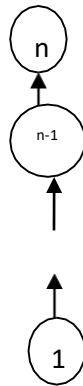
3. After performing union(1,2)operationParent[2]:=1

0	1	1	0	2	0
1	2	3	4	5	6



Process the following sequence of union operations $\text{Union}(1,2), \text{Union}(2,3), \dots, \text{Union}(n-1,n)$

Degenerate Tree:



The time taken for $n-1$ unions is $O(n)$.

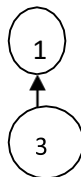
Find(i) operation: determines the root of the tree containing element i . Simple Algorithm for Find:

Algorithm Find(i)

```
{
    j:=i;
    while(p[j]>0) do
        j:=p[j]; return j;
}
```

Find Operation: Find(i) implies that it finds the root node of i^{th} node, in other words it returns the name of the set i .

Example: Consider the Union(1,3)

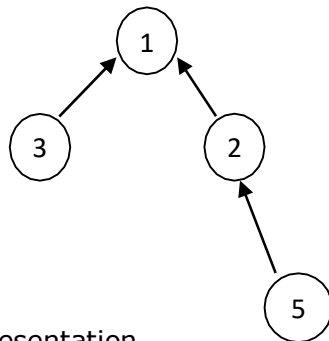


Find(1)=0

Find(3)=1, since its parent is 1. (i.e, root is 1)

Example:

Considering



Array Representation

P[i]	0	1	1	2
i	1	2	3	5

Find(5)=1

Find(2)=1

Find(3)=1

The root node represents all the nodes in the tree. Time Complexity of „n“ find operations is $O(n^2)$.

To improve the performance of union and find algorithms by avoiding the creation of degenerate tree. To accomplish this, we use weighting rule for Union(i,j).

Weighting Rule for Union(i,j)

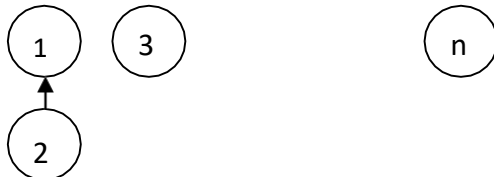
```
1  Algorithm WeightedUnion(i, j)
2  // Union sets with roots i and j,  $i \neq j$ , using the
3  // weighting rule.  $p[i] = -count[i]$  and  $p[j] = -count[j]$ .
4  {
5      temp := p[i] + p[j];
6      if (p[i] > p[j]) then
7          { // i has fewer nodes.
8              p[i] := j; p[j] := temp;
9          }
10     else
11         { // j has fewer or equal nodes.
12             p[j] := i; p[i] := temp;
13         }
14 }
```

Union algorithm with weighting rule

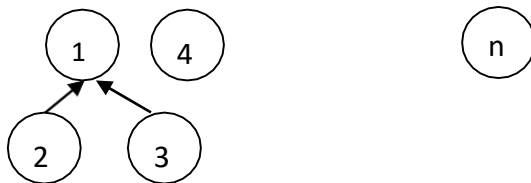
Tree obtained with
weighted Initially



Union(1,2)

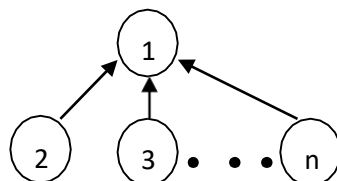


Union(1,3)



⋮
⋮
⋮

Union(1,n)



Collapsing Rule for Find(i)

```

1  Algorithm CollapsingFind( $i$ )
2  // Find the root of the tree containing element  $i$ . Use the
3  // collapsing rule to collapse all nodes from  $i$  to the root.
4  {
5       $r := i$ ;
6      while ( $p[r] > 0$ ) do  $r := p[r]$ ; // Find the root.
7      while ( $i \neq r$ ) do // Collapse nodes from  $i$  to root  $r$ .
8      {
9           $s := p[i]$ ;  $p[i] := r$ ;  $i := s$ ;
10     }
11     return  $r$ ;
12 }
```

Find algorithm with collapsing rule

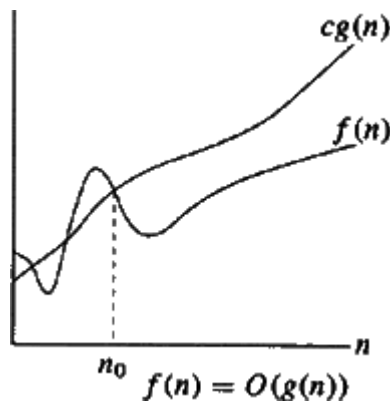
Asymptotic Notations:

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:

1. Big-OH(O)
2. Big-OMEGA(Ω),
3. Big-THETA (Θ)and
4. Little-OH(o)

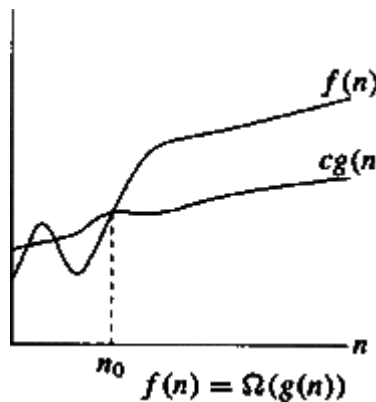
Big-OH O (Upper Bound)

$f(n) = O(g(n))$, (pronounced order of or big oh), says that the growth rate of $f(n)$ is less than or equal (\leq) that of $g(n)$.



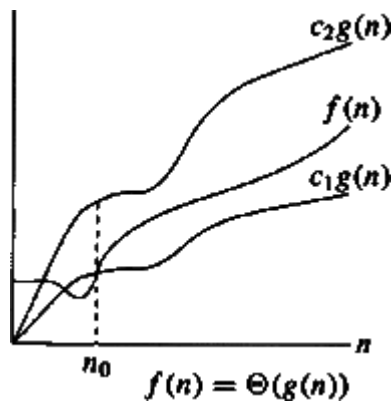
Big-OMEGA Ω (Lower Bound)

$f(n) = \Omega(g(n))$ (pronounced omega), says that the growth rate of $f(n)$ is greater than or equal (\geq) that of $g(n)$.



Big-THETA Θ (Same order)

$f(n) = \Theta(g(n))$ (pronounced theta), says that the growth rate of $f(n)$ equals (=) the growth rate of $g(n)$ [if $f(n) = O(g(n))$ and $T(n) = \Theta(g(n))$].



little-o notation

Definition: A theoretical measure of the execution of an *algorithm*, usually the time or memory needed, given the problem size n , which is usually the number of items. Informally, saying some equation $f(n) = o(g(n))$ means $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity. The notation is read, "f of n is little oh of g of n".

Formal Definition: $f(n) = o(g(n))$ means for all $c > 0$ there exists some $k > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq k$. The value of k must not depend on n , but may depend on c .

Different time complexities

Suppose „M“ is an algorithm, and suppose „n“ is the size of the input data. Clearly the complexity $f(n)$ of M increases as n increases. It is usually the rate of increase of $f(n)$ we want to examine. This is usually done by comparing $f(n)$ with some standard functions. The most common computing times are:

$$O(1), O(\log_2 n), O(n), O(n \cdot \log_2 n), O(n^2), O(n^3), O(2^n), n! \text{ and } n^n$$

Classification of Algorithms

If „n“ is the number of data items to be processed or degree of polynomial or the size of the file to be sorted or searched or the number of nodes in a graph etc.

- | | |
|-------------|--|
| 1 | Next instructions of most programs are executed once or at most only a few times. If all the instructions of a program have this property, we say that its running time is aconstant. |
| Logn | When the running time of a program is logarithmic, the program gets slightly slower as n grows. This running time commonly occurs in programs that solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction., When n is a million, $\log n$ is a doubled. Whenever n doubles, $\log n$ increases by a constant, but $\log n$ does not double until n increases to n^2 . |
| n | When the running time of a program is linear, it is generally the case that a |

small amount of processing is done on each input element. This is the optimal situation for an algorithm that must process n inputs.

- $n \log n$** This running time arises for algorithms that solve a problem by breaking it up into smaller sub-problems, solving them independently, and then combining the solutions. When n doubles, the running time more than doubles.
- n^2** When the running time of an algorithm is quadratic, it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop) whenever n doubles, the running time increases fourfold.
- n^3** Similarly, an algorithm that processes triples of data items (perhaps in a triple-nested loop) has a cubic running time and is practical for use only on small problems. Whenever n doubles, the running time increases eightfold.
- 2^n** Few algorithms with exponential running time are likely to be appropriate for practical use, such algorithms arise naturally as “brute-force” solutions to problems. Whenever n doubles, the running time squares.

Numerical Comparison of Different Algorithms

The execution time for six of the typical functions is given below:

n	$\log_2 n$	$n \cdot \log_2 n$	n^2	n^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65,536
32	5	160	1024	32,768	4,294,967,296
64	6	384	4096	2,62,144	Note 1
128	7	896	16,384	2,097,152	Note 2
256	8	2048	65,536	1,677,216	???????

Note1: The value here is approximately the number of machine instructions executed by a 1 gigaflop computer in 5000 years.