Algorithm specification

An algorithm is defined as a finite set of instructions that, if followed, performs a particular task. All algorithms must satisfy the following criteria

Input. An algorithm has zero or more inputs, taken or collected from a specified set of objects.

Output. An algorithm has one or more outputs having a specific relation to the inputs.

Definiteness. Each step must be clearly defined; Each instruction must be clear and unambiguous.

Finiteness. The algorithm must always finish or terminate after a finite number of steps.

Effectiveness. All operations to be accomplished must be sufficiently basic that they can be done exactly and in finite length.

We can depict an algorithm in many ways:

>>> Natural language: implement a natural language like English

>>> Flow charts: Graphic representations denoted flowcharts, only if the algorithm is small and simple.

>>> Pseudo code: This pseudo code skips most issues of ambiguity; no particularity regarding syntax programming language.

Example :Algorithm for calculating factorial value of a number

Step 1: a number n is inputted.

Step 2: variable final is set as 1.

Step 3: final<=final\*1.

Step 4: decrease n.

Step 5: verify if n is equal to zero.

Step 6: if n is equal to zero , goto Step 8(break out of loop).

Step 7: else goto step 3.

Step 8: the result final is printed.

Recursive Algorithm

A recursive algorithm calls itself which generally passes the return value as a parameter to the algorithm again. This parameter indicates the input while the return value indicates the output.

Recursive algorithm is defined as a method of simplification that divides the problem into sub-problems of the same nature. The result of one recursion is treated as the input for the next recursion. The repletion is in the self-similar fashion manner. The algorithm calls itself with smaller input values and obtains the results by simply accomplishing the operations on these smaller values. Generation of factorial, Fibonacci number series are denoted as the examples of recursive algorithms.

Example :Writing factorial function using recursion

intfactorialA(int n)

{

return n \* factorialA(n-1);

}

Performance Analysis

If we want to go from city "A" to city "B", there can be many ways of doing this. We can go by flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one which suits us. Similarly, in computer science, there are multiple algorithms to solve a problem. When we have more than one algorithm to solve a problem, we need to select the best one. Performance

Analysis helps us to select the best lgorithm from multiple algorithms to solve a problem. when there are multiple alternative algorithms to solve problem, we analyse them and pick the one which is best suitable for our requirements. The formal definition is as follows:

# Performance of an algorithm is a process of making evaluate judgement about algorithms.

It can also be defined as follows...

# Performance of an algorithm means predicting the resources which are required to an algorithm to peform its task.

Generally, the performance of an algorithm depends on the following elements...

1. Whether that algorithm is providing the exact solution for the problem?
2. Whether it is easy to understand?
3. Whether it is easy to implement?
4. How much space (memory) it requires to solve the problem?
5. How much time it takes to solve the problem?

When we want to analyse an algorithm, we consider only the space and time required by that particular algorithm and we ignore all the remaining elements.  
Based on this information, performance analysis of an algorithm can also be defined as follows...

# Performance analysis of an algorithm is the process of calculating space and time required by that algorithm.

Performance analysis of an algorithm is performed by using the following measures

# 1. Space required to complete the task of that algorithm (Space Complexity).

# 2. Time required to complete the task of that algorithm (Time Complexity).

R

Space complexity

When we design an algorithm to solve a problem, it needs some computer memory to complete its execution. For any algorithm, memory is required for the following purposes...

1. To store program instructions.
2. To store constant values.
3. To store variable values.
4. And for few other things like funcion calls, jumping statements etc,

Space complexity of an algorithm can be defined as follows...

# Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm.

Generally, when a program is under execution it uses the computer memory for three reasons they are as follows

Instruction space : it is the amount of memory used to store compiled version of instructions.

Environmental Stack: it is the amount of memory used to store information of partially executed functions at the time of function call.

Data Space: it is the amount of memory used to store all the variables and constants.

NOTE: when we want to perform analysis of an algorithm based on its space complexity we consider only data space and ignore Instruction space as well as Environmental stack that means we calculate only the memory required to store variables , constants, structures, etc

To calculate the space complexity we must know the memory required to store different datatype values according to the compiler.

1.2 bytes to store integer value.

2. 4 bytes to store floating point value.

3. 1 byte to store character value.

4. 6 or 8 bytes to store double value.

Example 1:

int square(int a)

{

return a\*a;

}

In the above example , it requires 2 bytes of memory to store variable a and another 2 bytes of memory is used for return value.

That means , totally it requires 4 bytes of memory to complete its execution and this 4 bytes of memory is fixed for any input value of a, this space complexity is said to be constance space complexity

## If any algorithm requires fixed amount of space for all input values then that space complexity is said to be constant space complexity

Example 2:

Int sum(int A[],int n)

{

int sum=0,i;

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

sum=sum+A[i];

retrun sum;

}

In the above example it requires n\*2 bytes of memory to store array variable a[]

2 bytes of memory for integer parameter ‘n’, 4 bytes of memory for local integer variable sum and i(2 bytes each),

2 bytes of memory for return value.

That means it requires 2n+8 bytes of memory to complete its execution. Here , the total amount of memory required depends on the value of n. As n value increase the space required also increase proportionately. This type of space complexity is said to be linear space complexity.

# If the amount of space required by an algorithm is increased with the increase of input value, then that space complexity is said to be linear space complexity.

Time complexity

Every algorithm requires some amount of computer time to execute its instruction to perform the task. This computer time required is called time complexity.  
The time complexity of an algorithm can be defined as follows...

# The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

Generally, the running time of an algorithm depends upon the following...

1. Whether it is running on **Single** processor machine or **Multi** processor machine.
2. Whether it is a **32 bit** machine or **64 bit** machine.
3. **Read** and **Write** speed of the machine.
4. The amount of time required by an algorithm to perform **Arithmetic** operations, **logical** operations, **return** value and **assignment** operations etc.,
5. **Input** data

**Note -**When we calculate time complexity of an algorithm, we consider only input data and ignore the remaining things, as they are machine dependent. We check only, how our program is behaving for the different input values to perform all the operations like Arithmetic, Logical, Return value and Assignment etc.,

Calculating Time Complexity of an algorithm based on the system configuration is a very difficult task because the configuration changes from one system to another system. To solve this problem, we must assume a model machine with a specific configuration. So that, we can able to calculate generalized time complexity according to that model machine.  
  
To calculate the time complexity of an algorithm, we need to define a model machine. Let us assume a machine with following configuration...

1. It is a Single processor machine
2. It is a 32 bit Operating System machine
3. It performs sequential execution
4. It requires 1 unit of time for Arithmetic and Logical operations
5. It requires 1 unit of time for Assignment and Return value
6. It requires 1 unit of time for Read and Write operations

Now, we calculate the time complexity of following example code by using the above-defined model machine...

Consider the following piece of code...

**Example 1**

int sum(int a, int b)

{

return a+b;

}

In the above sample code, it requires 1 unit of time to calculate a+b and 1 unit of time to return the value. That means, totally it takes 2 units of time to complete its execution. And it does not change based on the input values of a and b. That means for all input values, it requires the same amount of time i.e. 2 units.

# If any program requires a fixed amount of time for all input values then its time complexity is said to be constant time complexity.

Consider the following piece of code...

**Example 2**

int sum(int A[], int n)

{

int sum = 0, i;

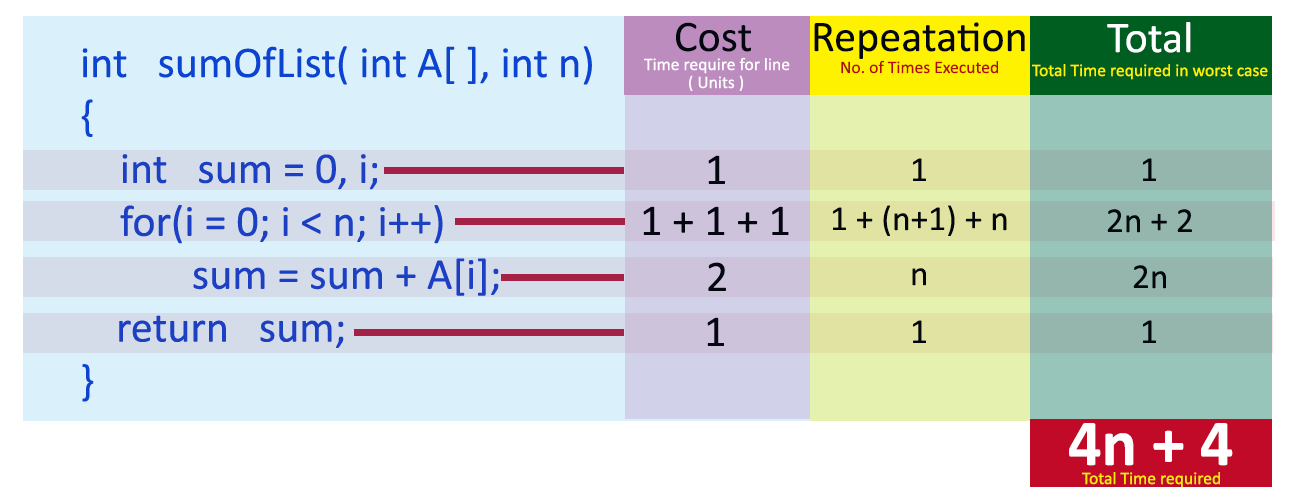
for(i = 0; i < n; i++)

sum = sum + A[i];

return sum;

}

For the above code, time complexity can be calculated as follows...



In above calculation cost is the amount of computer time required for a single operation in each line repeatation the amount of computer time required by each operation for all its repeatations total is the amount of computer time required by each operation to execute so above code requires 4n+4 units of computer the task .Here, the exact time is not fixed and it changes based on the value if we increase the n value then the time required also increases linearly

Totally it takes 4n+4units of time to complete its execution and it is linear time complexity

# If the amount of time required by an algorithm is increased with the increase of input value then that time complexity is said to be linear time complexity

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Asymptotic Analysis  As we know that data structure is a way of organizing the data efficiently and that efficiency is measured either in terms of time or space. So, the ideal data structure is a structure that occupies the least possible time to perform all its operation and the memory space. Our focus would be on finding the time complexity rather than space complexity, and by finding the time complexity, we can decide which data structure is the best for an algorithm.54.9M  Usually, the time required by an algorithm comes under three types:  **Worst case:** It defines the input for which the algorithm takes a huge time.  **Average case:** It takes average time for the program execution.  **Best case:** It defines the input for which the algorithm takes the lowest time Asymptotic Notations The commonly used asymptotic notations used for calculating the running time complexity of an algorithm is given below:   * Big oh Notation (O) * Omega Notation (Ω) * Theta Notation (θ)  Big oh Notation (O)  * Big O notation is an asymptotic notation that measures the performance of an algorithm by simply providing the order of growth of the function. * This notation provides an upper bound on a function which ensures that the function never grows faster than the upper bound. So, it gives the least upper bound on a function so that the function never grows faster than this upper bound.   It is the formal way to express the upper boundary of an algorithm running time. It measures the worst case of time complexity or the algorithm's longest amount of time to complete its operation. It is represented as shown below:  Asymptotic Analysis  **For example:**  If **f(n)** and **g(n)** are the two functions defined for positive integers,  then **f(n)** = **O(g(n))** as **f(n) is big oh of g(n)** or f(n) is on the order of g(n)) if there exists constants c and no such that:  **f(n)≤c.g(n) for all n≥no**  This implies that f(n) does not grow faster than g(n), or g(n) is an upper bound on the function f(n). In this case, we are calculating the growth rate of the function which eventually calculates the worst time complexity of a function, i.e., how worst an algorithm can perform.  **Let's understand through examples**  Example 1: f(n)=2n+3 , g(n)=n  Now, we have to find **Is f(n)=O(g(n))?**  To check f(n)=O(g(n)), it must satisfy the given condition:  **f(n)<=c.g(n)**  First, we will replace f(n) by 2n+3 and g(n) by n.  2n+3 <= c.n  Let's assume c=5, n=1 then  2\*1+3<=5\*1  5<=5  For n=1, the above condition is true.  If n=2  2\*2+3<=5\*2  7<=10  For n=2, the above condition is true.  We know that for any value of n, it will satisfy the above condition, i.e., 2n+3<=c.n. If the value of c is equal to 5, then it will satisfy the condition 2n+3<=c.n. We can take any value of n starting from 1, it will always satisfy. Therefore, we can say that for some constants c and for some constants n0, it will always satisfy 2n+3<=c.n. As it is satisfying the above condition, so f(n) is big oh of g(n) or we can say that f(n) grows linearly. Therefore, it concludes that c.g(n) is the upper bound of the f(n). It can be represented graphically as:  Asymptotic Analysis  The idea of using big o notation is to give an upper bound of a particular function, and eventually it leads to give a worst-time complexity. It provides an assurance that a particular function does not behave suddenly as a quadratic or a cubic fashion, it just behaves in a linear manner in a worst-case. Omega Notation (Ω)  * It basically describes the best-case scenario which is opposite to the big o notation. * It is the formal way to represent the lower bound of an algorithm's running time. It measures the best amount of time an algorithm can possibly take to complete or the best-case time complexity. * It determines what is the fastest time that an algorithm can run.   If we required that an algorithm takes at least certain amount of time without using an upper bound, we use big- Ω notation i.e. the Greek letter "omega". It is used to bound the growth of running time for large input size.  If **f(n)** and **g(n)** are the two functions defined for positive integers,  then **f(n) = Ω (g(n))** as **f(n) is Omega of g(n)** or f(n) is on the order of g(n)) if there exists constants c and no such that:  **f(n)>=c.g(n) for all n≥no and c>0**  **Let's consider a simple example.**  If f(n) = 2n+3, g(n) = n,  Is f(n)= **Ω** (g(n))?  It must satisfy the condition:  **f(n)>=c.g(n)**  To check the above condition, we first replace f(n) by 2n+3 and g(n) by n.  **2n+3>=c\*n**  Suppose c=1  **2n+3>=n** (This equation will be true for any value of n starting from 1).  Therefore, it is proved that g(n) is big omega of 2n+3 function.  Asymptotic Analysis  As we can see in the above figure that g(n) function is the lower bound of the f(n) function when the value of c is equal to 1. Therefore, this notation gives the fastest running time. But, we are not more interested in finding the fastest running time, we are interested in calculating the worst-case scenarios because we want to check our algorithm for larger input that what is the worst time that it will take so that we can take further decision in the further process. Theta Notation (θ)  * The theta notation mainly describes the average case scenarios. * It represents the realistic time complexity of an algorithm. Every time, an algorithm does not perform worst or best, in real-world problems, algorithms mainly fluctuate between the worst-case and best-case, and this gives us the average case of the algorithm. * Big theta is mainly used when the value of worst-case and the best-case is same. * It is the formal way to express both the upper bound and lower bound of an algorithm running time.   Let's understand the big theta notation mathematically:  Let f(n) and g(n) be the functions of n where n is the steps required to execute the program then:  **f(n)= θg(n)**  The above condition is satisfied only if when  **c1.g(n)<=f(n)<=c2.g(n)**  where the function is bounded by two limits, i.e., upper and lower limit, and f(n) comes in between. The condition **f(n)= θg(n)** will be true if and only if c1.g(n) is less than or equal to f(n) and c2.g(n) is greater than or equal to f(n). The graphical representation of theta notation is given below:  Asymptotic Analysis  Let's consider the same example where f(n)=2n+3 g(n)=n  As c1.g(n) should be less than f(n) so c1 has to be 1 whereas c2.g(n) should be greater than f(n) so c2 is equal to 5. The c1.g(n) is the lower limit of the of the f(n) while c2.g(n) is the upper limit of the f(n).  c1.g(n)<=f(n)<=c2.g(n)  Replace g(n) by n and f(n) by 2n+3  c1.n <=2n+3<=c2.n  if c1=1, c2=2, n=1  1\*1 <=2\*1+3 <=2\*1  **1** <= **5** <= **2** // for n=1, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n)  **If n=2**  1\*2<=2\*2+3<=2\*2  2<=7<=4 // for n=2, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n)  Therefore, we can say that for any value of n, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n). Hence, it is proved that f(n) is big theta of g(n). So, this is the average-case scenario which provides the realistic time complexity. Why we have three different asymptotic analysis? As we know that big omega is for the best case, big oh is for the worst case while big theta is for the average case. Now, we will find out the average, worst and the best case of the linear search algorithm.  Suppose we have an array of n numbers, and we want to find the particular element in an array using the linear search. In the linear search, every element is compared with the searched element on each iteration. Suppose, if the match is found in a first iteration only, then the best case would be Ω(1), if the element matches with the last element, i.e., nth element of the array then the worst case would be O(n). The average case is the mid of the best and the worst-case, so it becomes **θ(n/1). The constant terms can be ignored in the time complexity so average case would be θ(n)**.  So, three different analysis provide the proper bounding between the actual functions. Here, bounding means that we have upper as well as lower limit which assures that the algorithm will behave between these limits only, i.e., it will not go beyond these limits. Common Asymptotic Notations  |  |  |  | | --- | --- | --- | | constant | - | O(1) | | linear | - | O(n) | | logarithmic | - | O(log n) | | n log n | - | O(n log n) | | exponential | - | 2^O(n) | | cubic | - | O(n3) | | polynomial | - | n^O(1) | | quadratic | - | O(n2) |  | **Algorithm** | **Time Complexity** | | | Space Complexity | | --- | --- | --- | --- | --- | |  | **Best** | **Average** | **Worst** | Worst | | [Selection Sort](http://geeksquiz.com/selection-sort/) | Ω(n^2) | θ(n^2) | O(n^2) | O(1) | | [Bubble Sort](http://geeksquiz.com/bubble-sort/) | Ω(n) | θ(n^2) | O(n^2) | O(1) | | [Insertion Sort](http://geeksquiz.com/insertion-sort/) | Ω(n) | θ(n^2) | O(n^2) | O(1) | | [Heap Sort](http://geeksquiz.com/heap-sort/) | Ω(n log(n)) | θ(n log(n)) | O(n log(n)) | O(1) | | [Quick Sort](http://geeksquiz.com/quick-sort/) | Ω(n log(n)) | θ(n log(n)) | O(n^2) | O(log(n)) | | [Merge Sort](http://geeksquiz.com/merge-sort/) | Ω(n log(n)) | θ(n log(n)) | O(n log(n)) | O(n) | | [Bucket Sort](https://www.geeksforgeeks.org/bucket-sort-2/) | Ω(n +k) | θ(n +k) | O(n^2) | O(n) | | [Radix Sort](https://www.geeksforgeeks.org/radix-sort/) | Ω(nk) | θ(nk) | O(nk) | O(n + k) | | [Count Sort](https://www.geeksforgeeks.org/counting-sort/) | Ω(n +k) | θ(n +k) | O(n +k) | O(k) | | [Shell Sort](https://www.geeksforgeeks.org/shellsort/) | Ω(n) | θ(n log(n)) | O(n log(n)) | O(1) | | [Tim Sort](https://www.geeksforgeeks.org/timsort/) | Ω(n) | θ((n log(n))^2) | O((n log n (n))^2) | O(n) | | [Tree Sort](https://www.geeksforgeeks.org/tree-sort/) | Ω(n log(n)) | θ(n log(n)) | O(n^2) | O(n) | | [Cube Sort](https://www.geeksforgeeks.org/sort-the-array-according-to-their-cubes-of-each-element/) | Ω(n) | θ(n log(n)) | O(n log(n)) | O(n) |   Divide and Conquer Technique: Maximum Subarray Sum using Divide and Conquer algorithm You are given a one dimensional array that may contain both positive and negative integers, find the sum of contiguous subarray of numbers which has the largest sum.  For example, if the given array is {-2, -5, **6, -2, -3, 1, 5**, -6}, then the maximum subarray sum is 7 (see highlighted elements).  **The naive method**is to run two loops. The outer loop picks the beginning element, the inner loop finds the maximum possible sum with first element picked by outer loop and compares this maximum with the overall maximum. Finally return the overall maximum. The time complexity of the Naive method is O(n^2).  Using **Divide and Conquer** approach, we can find the maximum subarray sum in O(nLogn) time. Following is the Divide and Conquer algorithm.  **1)** Divide the given array in two halves **2)** Return the maximum of following three ….**a)** Maximum subarray sum in left half (Make a recursive call) ….**b)** Maximum subarray sum in right half (Make a recursive call) ….**c)** Maximum subarray sum such that the subarray crosses the midpoint  The lines 2.a and 2.b are simple recursive calls. How to find maximum subarray sum such that the subarray crosses the midpoint? We can easily find the crossing sum in linear time. The idea is simple, find the maximum sum starting from mid point and ending at some point on left of mid, then find the maximum sum starting from mid + 1 and ending with sum point on right of mid + 1. Finally, combine the two and return.   |  | | --- | | // A Divide and Conquer based program for maximum subarray sum problem  #include <stdio.h>  #include <limits.h>  // A utility funtion to find maximum of two integer  **int** max(**int** a, **int** b) { **return** (a > b)? a : b; }  // A utility funtion to find maximum of three integers  **int** max(**int** a, **int** b, **int** c) { **return** max(max(a, b), c); }  // Find the maximum possible sum in arr[] auch that arr[m] is part of it  **int** maxCrossingSum(**int** arr[], **int** l, **int** m, **int** h)  {      // Include elements on left of mid.  **int** sum = 0;  **int** left\_sum = INT\_MIN;  **for** (**int** i = m; i >= l; i--)      {          sum = sum + arr[i];  **if** (sum > left\_sum)            left\_sum = sum;      }        // Include elements on right of mid      sum = 0;  **int** right\_sum = INT\_MIN;  **for** (**int** i = m+1; i <= h; i++)      {          sum = sum + arr[i];  **if** (sum > right\_sum)            right\_sum = sum;      }        // Return sum of elements on left and right of mid  **return** left\_sum + right\_sum;  }    // Returns sum of maxium sum subarray in aa[l..h]  **int** maxSubArraySum(**int** arr[], **int** l, **int** h)  {     // Base Case: Only one element  **if** (l == h)  **return** arr[l];       // Find middle point  **int** m = (l + h)/2;       /\* Return maximum of following three possible cases        a) Maximum subarray sum in left half        b) Maximum subarray sum in right half        c) Maximum subarray sum such that the subarray crosses the midpoint \*/  **return** max(maxSubArraySum(arr, l, m),                maxSubArraySum(arr, m+1, h),                maxCrossingSum(arr, l, m, h));  }    /\*Driver program to test maxSubArraySum\*/  **int** main()  {  **int** arr[] = {2, 3, 4, 5, 7};  **int** n = **sizeof**(arr)/**sizeof**(arr[0]);  **int** max\_sum = maxSubArraySum(arr, 0, n-1);  **printf**("Maximum contiguous sum is %dn", max\_sum);  **getchar**();  **return** 0;  }  **Time Complexity :**  maxSubArraySum() is a recursive method and time complexity can be expressed as following recurrence relation. T(n)=2T(n/2)+Θ(n)The above recurrence is similar to [Merge Sort](http://geeksquiz.com/merge-sort/) and can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is Θ(nLogn).  [**The Kadane’s Algorithm**](https://tutorialspoint.dev/slugresolver/largest-sum-contiguous-subarray/)for this problem takes O(n) time. Therefore the Kadane’s algorithm is better than the Divide and Conquer approach, but this problem can be considered as a good example to show power of Divide and Conquer. The above simple approach where we divide the array in two halves, reduces the time complexity from O(n^2) to O(nLogn). |    Strassen’s Matrix Multiplication in algorithms **Introduction**  **Strassen** in 1969 which gives an overview that how we can find the multiplication of two **2\*2 dimension matrix by the brute-force algorithm**. But by using divide and conquer technique the overall complexity for multiplication two matrices is reduced. This happens by decreasing the total number if multiplication performed at the expenses of a slight increase in the number of addition.  For multiplying the two 2\*2 dimension matrices **Strassen's** used some formulas in which there are seven multiplication and eighteen addition, subtraction, and in brute force algorithm, there is eight multiplication and four addition. The utility of Strassen's formula is shown by its asymptotic superiority when order **n** of matrix reaches infinity. Let us consider two matrices **A** and **B**, **n\*n** dimension, where **n** is a power of two. It can be observed that we can contain four **n/2\*n/2** submatrices from **A**, **B** and their product **C**. **C** is the resultant matrix of **A** and **B**. Procedure of Strassen matrix multiplication There are some procedures:   1. Divide a matrix of order of 2\*2 recursively till we get the matrix of 2\*2. 2. Use the previous set of formulas to carry out 2\*2 matrix multiplication. 3. In this eight multiplication and four additions, subtraction are performed. 4. Combine the result of two matrixes to find the final product or final matrix.  Formulas for Stassen’s matrix multiplication In **Strassen’s matrix multiplication** there are seven multiplication and four addition, subtraction in total.  1. D1 = (a11 + a22) (b11 + b22)  2. D2 = (a21 + a22).b11  3. D3 = (b12 – b22).a11  4. D4 = (b21 – b11).a22  5. D5 = (a11 + a12).b22  6. D6 = (a21 – a11) . (b11 + b12)  7. D7 = (a12 – a22) . (b21 + b22)  C11 = d1 + d4 – d5 + d7  C12 = d3 + d5  C21 = d2 + d4  C22 = d1 + d3 – d2 – d6 Algorithm for Strassen’s matrix multiplication **Algorithm Strassen(n, a, b, d)**  begin  If n = threshold then compute  C = a \* b is a conventional matrix.  Else  Partition a into four sub matrices a11, a12, a21, a22.  Partition b into four sub matrices b11, b12, b21, b22.  Strassen ( n/2, a11 + a22, b11 + b22, d1)  Strassen ( n/2, a21 + a22, b11, d2)  Strassen ( n/2, a11, b12 – b22, d3)  Strassen ( n/2, a22, b21 – b11, d4)  Strassen ( n/2, a11 + a12, b22, d5)  Strassen (n/2, a21 – a11, b11 + b22, d6)  Strassen (n/2, a12 – a22, b21 + b22, d7)  C = d1+d4-d5+d7 d3+d5  d2+d4 d1+d3-d2-d6    end if    return (C)  end.  Code for strassen matrix multiplication  #include <stdio.h>  **int** main()  {  **int** a[2][2],b[2][2],c[2][2],i,j;  **int** m1,m2,m3,m4,m5,m6,m7;  printf("Enter the 4 elements of first matrix: ");  **for**(i=0;i<2;i++)  **for**(j=0;j<2;j++)  scanf("%d",&a[i][j]);  printf("Enter the 4 elements of second matrix: ");  **for**(i=0;i<2;i++)  **for**(j=0;j<2;j++)  scanf("%d",&b[i][j]);  printf("\nThe first matrix is\n");  **for**(i=0;i<2;i++)  {  printf("\n");  **for**(j=0;j<2;j++)  printf("%d\t",a[i][j]);  }  printf("\nThe second matrix is\n");  **for**(i=0;i<2;i++)  {  printf("\n");  **for**(j=0;j<2;j++)  printf("%d\t",b[i][j]);  }  m1= (a[0][0] + a[1][1])\*(b[0][0]+b[1][1]);  m2= (a[1][0]+a[1][1])\*b[0][0];  m3= a[0][0]\*(b[0][1]-b[1][1]);  m4= a[1][1]\*(b[1][0]-b[0][0]);  m5= (a[0][0]+a[0][1])\*b[1][1];  m6= (a[1][0]-a[0][0])\*(b[0][0]+b[0][1]);  m7= (a[0][1]-a[1][1])\*(b[1][0]+b[1][1]);  c[0][0]=m1+m4-m5+m7;  c[0][1]=m3+m5;  c[1][0]=m2+m4;  c[1][1]=m1-m2+m3+m6;  printf("\nAfter multiplication using \n");  **for**(i=0;i<2;i++)  {  printf("\n");  **for**(j=0;j<2;j++)  printf("%d\t",c[i][j]);  }  **return** 0;  }  **Output:**  Enter the 4 elements of first matrix:  5 6 1 7  Enter the 4 element of second matrix:  6 2 8 7  The first matrix is  5 6  1 7  The second matrix is  6 2  8 7  After multiplication  78 52  62 51  **Complexity:** The time complexity is **O(N2.8074)**. aRecurrence Relation A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs. To solve a Recurrence Relation means to obtain a function defined on the natural numbers that satisfy the recurrence.  **For Example**, the Worst Case Running Time T(n) of the MERGE SORT Procedures is described by the recurrence.  T (n) = θ (1) if n=1  2TDAA Recurrence Relation + θ (n) if n>1  There are four methods for solving Recurrence:   1. [Substitution Method](https://www.javatpoint.com/daa-recurrence-relation" \l "substitution-method) 2. [Iteration Method](https://www.javatpoint.com/daa-recurrence-relation" \l "iteration-method) 3. [Recursion Tree Method](https://www.javatpoint.com/daa-recursion-tree-method)   4.Master Method 1. Substitution Method: The Substitution Method Consists of two main steps:  54.5M  1.1K  Exception Handling in Java - Javatpoint   1. Guess the Solution. 2. Use the mathematical induction to find the boundary condition and shows that the guess is correct.   **For Example1** Solve the equation by Substitution Method.  T (n) = TDAA Recurrence Relation + n  We have to show that it is asymptotically bound by O (log n).  **Solution:**  For T (n) = O (log n)  We have to show that for some constant c   1. T (n) ≤c logn.   Put this in given Recurrence Equation.  T (n) ≤c logDAA Recurrence Relation+ 1  ≤c logDAA Recurrence Relation+ 1 = c logn-clog2 2+1  ≤c logn for c≥1  Thus **T (n) =O logn**.  **Example2** Consider the Recurrence  T (n) = 2TDAA Recurrence Relation+ n n>1  Find an Asymptotic bound on T.  **Solution:**  We guess the solution is O (n (logn)).Thus for constant 'c'.  T (n) ≤c n logn  Put this in given Recurrence Equation.  Now,  T (n) ≤2cDAA Recurrence Relationlog DAA Recurrence Relation+n  ≤cnlogn-cnlog2+n  =cn logn-n (clog2-1)  ≤cn logn for (c≥1)  Thus **T (n) = O (n logn)**. 2. Iteration Methods It means to expand the recurrence and express it as a summation of terms of n and initial condition.  **Example1:** Consider the Recurrence   1. T (n) = 1  **if** n=1 2. = 2T (n-1) **if** n>1   **Solution:**    T (n) = 2T (n-1)  = 2[2T (n-2)] = 22T (n-2)  = 4[2T (n-3)] = 23T (n-3)  = 8[2T (n-4)] = 24T (n-4) (Eq.1)  Repeat the procedure for i times  T (n) = 2i T (n-i)  Put n-i=1 or i= n-1 in (Eq.1)  T (n) = 2n-1 T (1)  = 2n-1 .1 {T (1) =1 .....given}  = 2n-1  **Example2:** Consider the Recurrence   1. T (n) = T (n-1) +1 and T (1) =  θ (1).   **Solution:**  T (n) = T (n-1) +1  = (T (n-2) +1) +1 = (T (n-3) +1) +1+1  = T (n-4) +4 = T (n-5) +1+4  = T (n-5) +5= T (n-k) + k  Where k = n-1  T (n-k) = T (1) = θ (1)  T (n) = θ (1) + (n-1) = 1+n-1=n= θ (n). Recursion Tree Method 1. Recursion Tree Method is a pictorial representation of an iteration method which is in the form of a tree where at each level nodes are expanded.  2. In general, we consider the second term in recurrence as root.  3. It is useful when the divide & Conquer algorithm is used.  4. It is sometimes difficult to come up with a good guess. In Recursion tree, each root and child represents the cost of a single subproblem.  5. We sum the costs within each of the levels of the tree to obtain a set of pre-level costs and then sum all pre-level costs to determine the total cost of all levels of the recursion.  6. A Recursion Tree is best used to generate a good guess, which can be verified by the Substitution Method.  **Example 1**  Consider T (n) = 2TDAA Recurrence Relation + n2  We have to obtain the asymptotic bound using recursion tree method.  **Solution:** The Recursion tree for the above recurrence is    **Example 2:** Consider the following recurrence  T (n) = 4TDAA Recurrence Relation +n  Obtain the asymptotic bound using recursion tree method.  **Solution:** The recursion trees for the above recurrence  DAA Recursion Tree Method 6 2 8 7 |

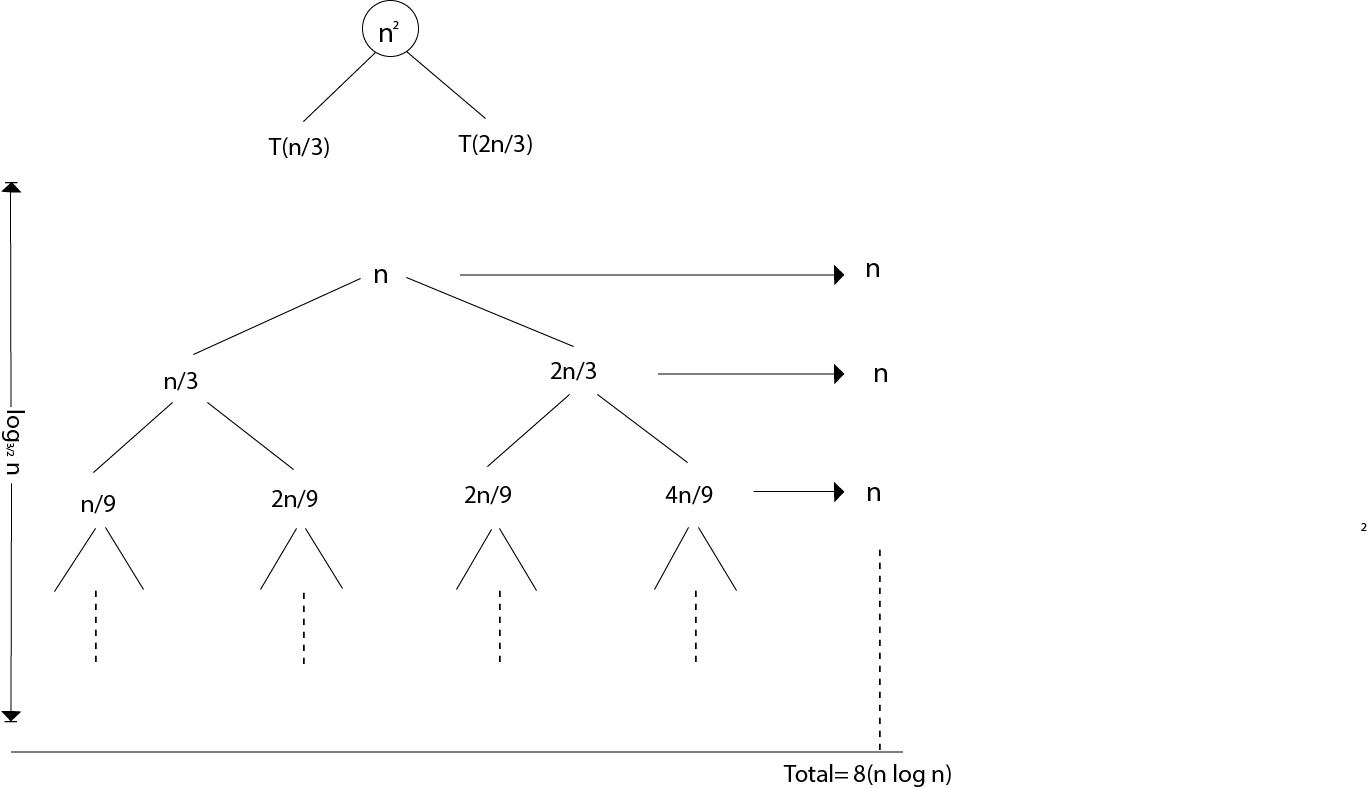
Hgftfajysksgygjshbgfhsf

**Example 3:** Consider the following recurrence

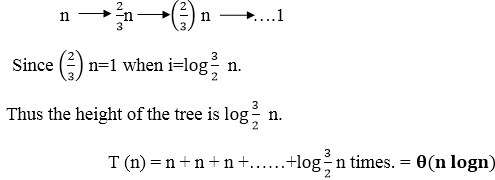
DAA Recursion Tree Method

Obtain the asymptotic bound using recursion tree method.

**Solution:** The given Recurrence has the following recursion tree



When we add the values across the levels of the recursion trees, we get a value of n for every level. The longest path from the root to leaf is



Ggtghhkjgytddtygho

Master Method

The Master Method is used for solving the following types of recurrence

T (n) = a TDAA Master Method+ f (n) with a≥1 and b≥1 be constant & f(n) be a function and DAA Master Methodcan be interpreted as

Let T (n) is defined on non-negative integers by the recurrence.

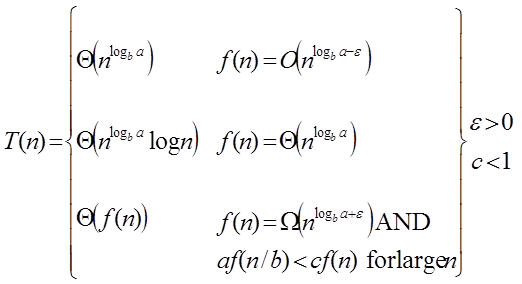
T (n) = a TDAA Master Method+ f (n)

In the function 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 sum of the work done outside the recursive calls, which includes the sum of dividing the problem and the sum of combining the solutions to the subproblems.
* It is not possible always bound the function according to the requirement, so we make three cases which will tell us what kind of bound we can apply on the function.

Master Theorem:

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



**Case1:** If f (n) = DAA Master Method for some constant ε >0, then it follows that:

T (n) = Θ DAA Master Method

**Example:**

T (n) = 8 T DAA Master Method apply master theorem on it.

**Solution:**

Compare T (n) = 8 T DAA Master Method with

T (n) = a T DAA Master Method

a = 8, b=2, f (n) = 1000 n2, logba = log28 = 3

Put all the values in: f (n) = DAA Master Method

1000 n2 = O (n3-ε )

If we choose ε=1, we get: 1000 n2 = O (n3-1) = O (n2)

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

T (n) = Θ DAA Master Method

Therefore: T (n) = Θ (n3)

**Case 2:** If it is true, for some constant k ≥ 0 that:

F (n) = Θ DAA Master Method then it follows that: T (n) = Θ DAA Master Method

**Example:**

T (n) = 2 DAA Master Method, solve the recurrence by using the master method.

As compare the given problem with T (n) = a TDAA Master Method a = 2, b=2, k=0, f (n) = 10n, logba = log22 =1

Put all the values in f (n) =Θ DAA Master Method, we will get

10n = Θ (n1) = Θ (n) which is true.

**Therefore:** T (n) = Θ DAA Master Method

= Θ (n log n)

**Case 3:** If it is true f(n) = Ω DAA Master Method for some constant ε >0 and it also true that: a f DAA Master Method for some constant c<1 for large value of n ,then :

1. T (n) = Θ((f (n))

**Example:** Solve the recurrence relation:

T (n) = 2 DAA Master Method

**Solution:**

Compare the given problem with T (n) = a T DAA Master Method

a= 2, b =2, f (n) = n2, logba = log22 =1

Put all the values in f (n) = Ω DAA Master Method ..... (Eq. 1)

If we insert all the value in (Eq.1), we will get

n2 = Ω(n1+ε) put ε =1, then the equality will hold.

n2 = Ω(n1+1) = Ω(n2)

Now we will also check the second condition:

2 DAA Master Method

If we will choose c =1/2, it is true:

DAA Master Method ∀ n ≥1

So it follows: T (n) = Θ ((f (n))

T (n) = Θ(n2)