Design and analysis of an algorithms
- Notes by Sivarajan R

Logarithmic rule chart

Rule 1: $\log_b(M \cdot N) = \log_b M + \log_b N$

Rule 2: $\log_b\left(\frac{M}{N}\right) = \log_b M - \log_b N$

Rule 3: $\log_b\left(M^k\right) = k \cdot \log_b M$

Rule 4: $\log_b(1) = 0$

Rule 5: $\log_b(b) = 1$

Rule 6: $\log_b\left(b^k\right) = k$

Rule 7: $b^{\log_b(k)} = k$

Where:

$b > 0$ but $b \neq 1$, and M, N, and k are real numbers but M and N must be positive!

Sequence and Series formula

Arithmetic Progression:
5,10,15...n
General term t_n= a (n -1)+ d
where:
a: start term (5)
common difference d = Difference of any 2 consecutive no. = 10 -5 = 5

Geometric progression

2^1,2^2,2^3...2^n
General term t_n= ar^{n-1}
where:
a: start term (5)

r: Rate of growth = Division of any 2 consecutive no. =

$2^3 / 2^2$ = 2

Sum of Arithematic Series $= \frac{n}{2}[2a(n-1)d]$ (or) $\frac{n}{2}[\text{first term} + \text{last term}]$

Sum of Gemoetric series $= a\frac{(r^n - 1)}{(r-1)}$ if r != 1

Sum of Gemoetric series (if r = 1) $= na$

Sum of Gemoetric series (if -1 < r < 1) $= \frac{a}{(1-r)}$

Special Series

$1+2+3 \ldots n = \frac{(n(n+1))}{2}$

$1^2+2^2+3^2 \ldots n = \frac{(n(n+1)(2n+1))}{6}$

$1^3+2^3+3^3 \ldots n^3 = \{\frac{(n(n+1))}{2}\}^2$

sum of odd numbers $= n^2$

sum of odd numbers with last term L $= (\frac{(L+1)}{2})^2$

$\sum_{j=l}^{n}(k) = (n-l)k$

Reference area. This will help you to analyze some of the algorithms presented in this paper and also help in deep dive of the subject.

The reference area sums are mostly based on this playlist Credits: Abdul Bari YouTube channel.

I) Time complexity of for loops

```
1) for i=0 to n  step i = i +1 (or i = i -1) //executes n+1 times
    print(i) //executes n times
end for
```

time complexity is o(N)

2)for i=0 to n step i = i +2 (or i = i -1) //executes n+1 times
    print(i) //executes n /2times
end for

time complexity is o(N)
note: only take the part with N not the constants

3) for i=0 to n step i = i +1 (or i = i -1)
        for j=0 to i step j =j +1
            print(i) //executes n times
        end for
   end for

The sample execution is given below:

i	j	no of times of execution
0	(0)	0
1	0 (1)	1
2	0 1 (2)	2
3	0 1 2 (3)	3
n	0 . . n	n

Considering red colored(3rd one) column let us calculate the time complexity of the equation:

$1+2+3+4+5... n = (n(n+1))/2 = (n^2+1)/2$
take only n^2
therefore $o(n^2)$ is the time complexity

```
4)for i=0 ; p<= n  step i = i +1
              p = p+1
   end for
```

Note: the algorithm repeats when p<=n so therefore it is not running n times. it is running k times.

i	p
1	1
2	1+2
3	1+2+3
4	1+2+3+4
k	1+2+3 ... k

Assume:
p > n
p = (k(k+1))/2

(k(k+1))/2 > n
we assume k^2 roughly

$k^2 > n$

$k > o(n^{1/2})$

(square root of n)

```
5) for i=0 ; i < n  step i = i * 2
    print(i)
end for
    Tracing out i value
```

note: that the i multiplies until it is equal to n so that it has to multiply k times

i
1 * 2 = 2
2 *2 = 2^2
2^3
2^4
...
2^k

Assume i >= n

$i = 2^k$
$2^k = n$

therefore
$k = \log_2 n$
time complexity is o(log N) ($\log_2 n$ is also denoted as log n)

Important points:
- log n can give float value
- in that case you can get the ceil value of the answer
- eg : ceil of 2.2 is 3 (take the smallest next integer greater than 2.2)

6)
```
for i=n ; i >= 1  step i = i / 2
    print(i) //executes n times
end for
```

i
n
n/2
$n/2^2$

$n/2^3$
$n/2^4$
...
$n/2^k$

Assume i <1

$n/2^k < 1$
$2^k = n$

therefore
$k = \log_2 n$

7)

```
for i=0 to n  step i = i +1
    print(i)  // runs n times
end for

 for j=0 to n  step j =j +1
    print(j) // runs n times
end for
```

there fore time complexity will be: n+n=2n = n (only take the part with n no need coefficients)

8)

```
 for i=0 ; i < n  step i = i * 2
    p++ // runs log n
end for

for j=0 ; j<= p  step j = j* 2
    print(j) // runs for log p times since it runs until j
    <p (refer 5th case)

end for
```

from the above W.K.T
p = log n

therefore log (log n) is time complexity of this.

9)
```
for i=0 ; i < n  step i = i++
//takes n time to execute
    for j=0 ; j<n  step j = j* 2
    // takes n x log n time to execute
        print(j) // takes n x log n time to  execute

    end for
end for
```
therefore time complexity is n log n (add up all and do not consider coefficients of n)

10)
```
 for i=0 ; i < n  step i = i * 3
    print(i)
end for
```

i
1 * 3 = 3
3 *3 = 3^3
3^3
3^4
3^5
...
3^k

Assume i >= n

3^k =n

therefore

$k = \log_3 n$
(just for your knowledge)

Definition of Algorithm: An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the
input into the output.

Analysis of Algorithm:

Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis –

- **Worst-case** – The maximum number of steps taken on any instance of size **a**.
- **Best-case** – The minimum number of steps taken on any instance of size **a**.
- **Average case** – An average number of steps taken on any instance of size **a**.
- **Amortized** – A sequence of operations applied to the input of size **a** averaged over time.

Dive into basics of insertion sort:

Insertion sort is a very simple method to sort numbers in an ascending or descending order. This method follows the

incremental method. It can be compared with the technique how cards are sorted at the time of playing a game. The numbers, which are needed to be sorted, are known as **keys**. Here is the algorithm of the insertion sort method.

Algorithm: Insertion-Sort(A)

```
1. for j = 2 to A.length
2.   key = A[j]
3.   i = j – 1
4.   //Insert A[ j ] into the sorted sequence
5.    while i > 0 and A[i] > key
6.        A[i + 1] = A[i]
7.   i = i -1
8.   A[i + 1] = key
```

We use loop in-variants to help us understand why an algorithm is correct. We
must show three things about a loop invariant:

- **Initialization:** It is true prior to the first iteration of the loop.
- **Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.
- **Termination:** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

For Insertion sort:
Initialization: We start by showing that the loop invariant holds before the first loop iteration, when j = 2. 1 The sub-array A[1 . . j – 1], therefore, consists of just the single element A[1], which is in fact the original element

in A[1]. Moreover, this sub-array is sorted (trivially, of course), which shows that the loop invariant holds prior to the first iteration of the loop.

in short
Prior to the loop j =2 -> A[1.. j-1] = A[1] which contains only the A[1.. j-1] elements (of which there is only one) and since there is only a single element they are trivially sorted.

Maintenance: Next, we tackle the second property: showing that each iteration maintains the loop invariant. Informally, the body of the outer for loop works by moving A[j − 1], A[j − 2], A[j − 3], and so on by one position to the right until the proper position for A[j] is found (lines 4 7), at which point the value of A[j] is inserted (line 8). A more formal treatment of the second property would require us to state and show a loop invariant for the "inner" while loop. this point, however, we prefer not to get bogged down in such formalism,and so we rely on our informal analysis to show that the second property holds for the outer loop.

in short:

The outer **for** loop selects element A[j] and positions it properly into A[1.. j-1] via the while loop. Since the array A[1.. j-1] began sorted, inserting element A[j] into the proper place produces A[1.. j] in sorted order (and contains the first j elements).

Termination:

Finally, we examine what happens when the loop terminates. For insertion sort, the outer for loop ends when j exceeds n, i.e., when j = n + 1. Substituting n + 1 for j in the

wording of loop invariant, we have that the sub array A[1 . . n] consists of the elements originally in A[1 . . n], but in sorted

in other words:
The loop terminates when $j = n+1 \Rightarrow A[1..\ j-1] = A[1..\ (n+1)-1] = A[1..\ n]$ which since the array remains sorted after each iteration gives $A[1..\ n]$ is sorted when the loop terminates (and contains *all* the original elements) ⇒ the entire *original* array is sorted.

Analysis
Run time of this algorithm is very much dependent on the given input.

If the given numbers are sorted, this algorithm runs in ***O(n)*** time. If the given numbers are in reverse order, the algorithm runs in $\boldsymbol{O(n^2)}$ time.

Lets see how to derive those here

For all analysis in this course we assume that the algorithm will be implemented by a *program* run on a *generic computer*, i.e. single processor with random access memory (parallel algorithms are beyond the scope of this course but are extremely important in today's computing environments).

We will define the *input size*, *n*, to typically be the number of elements in the input set (but it could represent the number of bits in a representation or other appropriate enumeration). The *running time* will then be the total number of execution steps as a function of *n* the algorithm takes to complete. We will assume that each line of pseudo

code executes in a *constant* amount of time (although that amount may vary from line to line). Thus we multiply the (constant) time each line takes to execute by the number of times the line executes to find a cost per line and then sum the costs of all lines to give the *run time* of the algorithm.

For insertion sort we will define a variable t_j to be the number of times the while loop test is performed (which will be one more time than the body of the loop is executed) where j =2,3,..., n with n = *A.length*.

Statement	Cost	Times	Reason
1. for j = 2 to A.length	C_1	n	Since comparison occurs n times
2. key = A[j]	C_2	n – 1	The loop exits at nth comparison so it runs n -1
3. i = j – 1	C_3	n – 1	,,
4. //Insert A[j]into the sorted sequence	C_4	n – 1	Comment has cost of running 0
5.while i > 0 and A[i] >key	C_5	$\sum_{j=2}^{n}(T_j)$	Amount of comparison
6. A[i + 1] = A[i]	C_6	$\sum_{j=2}^{n}(T_j-1)$	Amount of swaps
7. i = i -1	C_7	$\sum_{j=2}^{n}(T_j-1)$	,,
8. A[i + 1] = key	C_8	n – 1	Same as 2

General Run Time

The run time for insertion sort can then be written as

$$T(n)=C_1 n+(C_2(n-1)+C_3(n-1)+C_4(n-1)+C_8(n-1))+(C_6\sum_{j=2}^{n}(T_j-1)+C_7\sum_{j=2}^{n}(T_j-1))+C_5\sum_{j=2}^{n}(T_j)$$

where the c_i's are the cost of each line (noting that c_3 = 0 since line 3 is a comment. It can also be omitted).

Best case:

- The best case for insertion sort is when the input array is already sorted, in which case the while loop never executes (but the condition must be checked once).
- Note that in best case minium one amount of comparison takes place and 0 swaps take place since the array is already sorted.
- Thus t_j = 1 (Comparison) for all j =2,3,... n (i.e. t_j-1 = 0(Swaps)) and the run time reduces to:

$$T(n)=C_1 n+(C_2(n-1)+C_3(n-1)+C_4(n-1)+C_8(n-1))+(C_6\sum_{j=2}^{n}(T_j-1)+C_7\sum_{j=2}^{n}(T_j-1))+C_5\sum_{j=2}^{n}(T_j)$$

Substuting $T_j=1$

$$T(n)=C_1 n+(C_2(n-1)+C_4(n-1)+0+C_8(n-1))+(C_6\sum_{j=2}^{n}0)+C_7\sum_{j=2}^{n}0+C_5\sum_{j=2}^{n}(1)$$

$$=C_1 n+C_2 n+C_2+C_3 n+C_3+C_8 n+C_8+C_6*0+C_7*0+C_5 n$$

$$=(C_1+C_2+C_4+C_5+C_8)n-(C_2+C_4+C_5+C_8)$$

$$=an+b$$

It is in form of linear equation
therefore the time complexity will be O(n) in best case

Note:

Case 2: Worst Case

The worst case for insertion sort is when the input array is in reverse (decreasing) sorted order, in which case the while loop executes the maximum number of times. Thus $t_j = j$ for $j = 2,3,\ldots, n$. Using Appendix A of CLRS, the summation terms can be reduced as follows:

Substuting $T_j = n$

$$T(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_8(n-1) + C_6\sum_{j=2}^{n}(n-1) + C_7\sum_{j=2}^{n}(n-1) + C_5\sum_{j=2}^{n}(n)$$

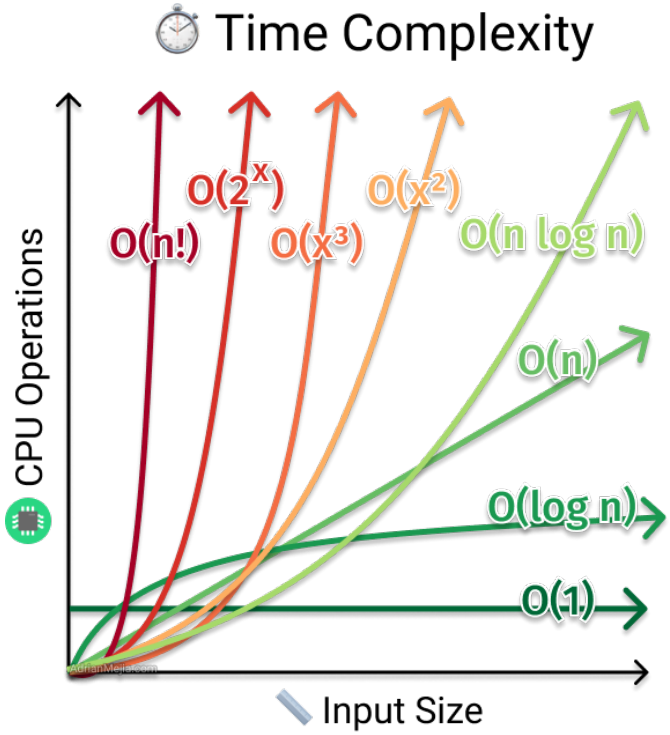
$$= C_1 n + C_2(n-1) + C_3(n-1) + C_8(n-1) + C_6\frac{(n(n-1))}{2} + C_7\frac{(n(n-1))}{2} + C_5\frac{(n(n+1))}{2}$$

Expanding this and grouping like terms we will get this

$$= \left(\frac{C_5}{2} + \frac{C_6}{2} + \frac{C_7}{2}\right)n^2 + \left(\frac{C_5}{2} + \frac{C_6}{2} + \frac{C_7}{2} + C_1 + C_2 + C_4 + C_8\right)n + \left(C_1 + C_2 + C_4 + C_{\%} + C_8\right)$$

therefore time complexity is O(n²)

Different time complexity comparison



Asymptotic notation

The complexity of an algorithm describes the efficiency of the algorithm in terms of the amount of the memory required to process the data and the processing time.

Complexity of an algorithm is analyzed in two perspectives: **Time** and **Space**.

Time Complexity

It’s a function describing the amount of time required to run an algorithm in terms of the size of the input. "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit

related to the amount of real time the algorithm will take.

Space Complexity

It’s a function describing the amount of memory an algorithm takes in terms of the size of input to the algorithm. We often speak of "extra" memory needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed-length) units to measure this.

Space complexity is sometimes ignored because the space used is minimal and/or obvious, however sometimes it becomes as important an issue as time.

Asymptotic Notations

What is Asymptotic Behavior

The word **Asymptotic** means approaching a value or curve arbitrarily closely (i.e., as some sort of limit is taken).

The only difference being, here we do not have to find the value of any expression where n is approaching any finite number or infinity, but in case of Asymptotic notations, we use the same model to ignore the constant factors and insignificant parts of an expression, to device a better way of representing complexities of algorithms, in a single coefficient, so that comparison between algorithms can be done easily.

Execution time of an algorithm depends on the instruction set, processor speed, disk I/O speed, etc. Hence, we

estimate the efficiency of an algorithm asymptotically.

Time function of an algorithm is represented by **T(n)**, where **n** is the input size.

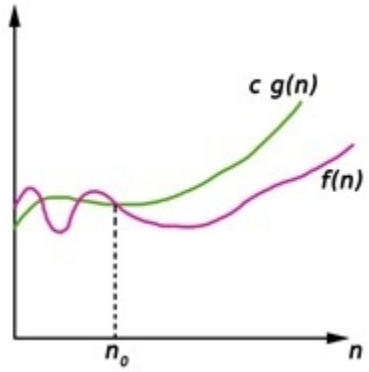
Different types of asymptotic notations are used to represent the complexity of an algorithm. Following asymptotic notations are used to calculate the running time complexity of an algorithm.

- **O** – Big Oh
- **Ω** – Big omega
- **θ** – Big theta

O: Asymptotic Upper Bound

'O' (Big Oh) is the most commonly used notation. A function ***f(n)*** can be represented is the order of ***g(n)*** that is ***O(g(n))***, if there exists a value of positive integer **n** as $\mathbf{n_0}$ and a positive constant **c** such that −

$O(f(n)) = \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } f(n) \leq c.g(n) \text{ for all } n > n_0. \}$



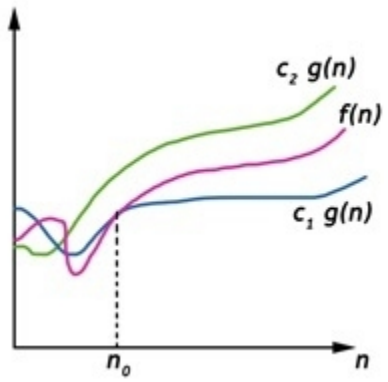
Hence, function ***g(n)*** is an upper bound for function ***f(n)***, as ***g(n)*** grows faster than ***f(n)***.

Big Omega Notation

Big-Omega (Ω) notation gives a lower bound for a function f(n) to within a constant factor.

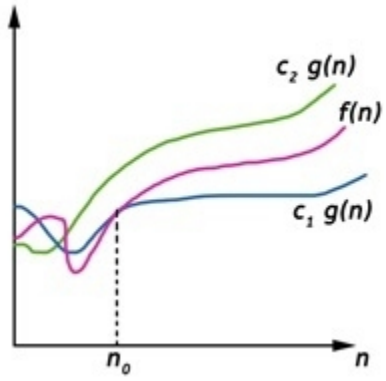
We write f(n) = Ω(g(n)), If there are positive constantsn0 and c such that, to the right of n0 the f(n) always lies on or above c*g(n).

Ω(g(n)) = { f(n) : There exist positive constant c and n0 such that 0 ≤ c g(n) ≤ f(n), for all n ≥ n0}



Big Theta Notation

Big-Theta(0) notation gives bound for a function f(n) to within a constant factor.



We write f(n) = 0(g(n)), If there are positive constants n0 and c1 and c2 such that, to the right of n0 the f(n) always lies between c1*g(n) and c2*g(n) inclusive.

0(g(n)) = {f(n) : There exist positive constant c1, c2 and n0 such that 0 ≤ c1 g(n) ≤ f(n) ≤ c2 g(n), for all n ≥ n0}

Note: any symbol from the above can be used for any cases.

Analysis Types:

Worst Case Analysis (Usually Done)

In the worst case analysis, we calculate upper bound on the running time of an algorithm. We must know the case that causes the maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be 0(n).

Average Case Analysis (Sometimes done)

In average case analysis, we take all possible inputs and

calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by a total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed(including the case of x not being present in array). So we sum all the cases and divide the sum by (n+1). Following is the value of average case time complexity.

Best Case Analysis (Bogus)
In the best case analysis, we calculate lower bound on the running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be 0(1)

Just for knowledge This can be skipped:

Assuming f(n), g(n) and h(n) be asymptotic functions the mathematical definitions are:

1. If **f(n) = 0(g(n))**, then there exists positive constants c1, c2, n0 such that **0 ≤ c1.g(n) ≤ f(n) ≤ c2.g(n)**, for all n ≥ n0
2. If **f(n) = O(g(n))**, then there exists positive constants c, n0 such that **0 ≤ f(n) ≤ c.g(n)**, for all n ≥ n0
3. If **f(n) = Ω(g(n))**, then there exists positive constants c, n0 such that **0 ≤ c.g(n) ≤ f(n)**, for all n ≥ n0
4. If **f(n) = o(g(n))**, then there exists positive constants c, n0 such that **0 ≤ f(n) < c.g(n)**, for all n ≥ n0
5. If **f(n) = ω(g(n))**, then there exists positive constants

c, n0 such that **0 ≤ c.g(n) < f(n)**, for all n ≥ n0

Properties:

1. **Reflexivity:**
   If f(n) is given then

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

   Example:

   If $f(n) = n^3 \Rightarrow O(n^3)$
   Similarly,

   $f(n) = \Omega(f(n))$
   $f(n) = \theta(f(n))$

2. **Symmetry:**

   f(n) = θ(g(n)) if and only if g(n) = θ(f(n))

   Example:

   If $f(n) = n^2$ and $g(n) = n^2$ then $f(n) = \theta(n^2)$ and $g(n) = \theta(n^2)$

   Proof:

   - ***Necessary part:***
     $f(n) = \theta(g(n)) \Rightarrow g(n) = \theta(f(n))$
     By the definition of θ, there exists positive constants c1, c2, no such that $c1.g(n) \leq f(n) \leq c2.g(n)$ for all $n \geq no$
     $\Rightarrow g(n) \leq (1/c1).f(n)$ and $g(n) \geq (1/c2).f(n)$
     $\Rightarrow (1/c2).f(n) \leq g(n) \leq (1/c1).f(n)$
     Since c1 and c2 are positive constants, 1/c1 and 1/c2 are well defined. Therefore, by the definition of θ, g(n) = θ(f(n))
   - ***Sufficiency part:***

$g(n) = \theta(f(n)) \Rightarrow f(n) = \theta(g(n))$
By the definition of θ, there exists positive constants c1, c2, no such that $c1.f(n) \leq g(n) \leq c2.f(n)$ for all $n \geq no$
$\Rightarrow f(n) \leq (1/c1).g(n)$ and $f(n) \geq (1/c2).g(n)$
$\Rightarrow (1/c2).g(n) \leq f(n) \leq (1/c1).g(n)$
By the definition of Theta(θ), $f(n) = \theta(g(n))$

3. **Transitivity:**

$f(n) = O(g(n))$ and $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$

Example:

If $f(n) = n$, $g(n) = n^2$ and $h(n) = n^3$
$\Rightarrow$ n is $O(n^2)$ and n^2 is $O(n^3)$ then n is $O(n^3)$

Proof:

$f(n) = O(g(n))$ and $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
By the definition of Big-Oh(O), there exists positive constants c, no such that $f(n) \leq c.g(n)$ for all $n \geq no$
$\Rightarrow f(n) \leq c1.g(n)$
$\Rightarrow g(n) \leq c2.h(n)$
$\Rightarrow f(n) \leq c1.c2h(n)$
$\Rightarrow f(n) \leq c.h(n)$, where, $c = c1.c2$ By the definition, $f(n) = O(h(n))$
Similarly,

$f(n) = \theta(g(n))$ and $g(n) = \theta(h(n)) \Rightarrow f(n) = \theta(h(n))$
$f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$
$f(n) = o(g(n))$ and $g(n) = o(h(n)) \Rightarrow f(n) = o(h(n))$
$f(n) = \omega(g(n))$ and $g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$

4. **Transpose Symmetry:**

$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

Example:

If f(n) = n and g(n) = n^2 then n is $O(n^2)$ and n^2 is Ω(n)

Proof:

- ***Necessary part:***
  f(n) = O(g(n)) ⇒ g(n) = Ω(f(n))
  By the definition of Big-Oh (O) ⇒ f(n) ≤ c.g(n) for some positive constant c ⇒ g(n) ≥ (1/c).f(n)
  By the definition of Omega (Ω), g(n) = Ω(f(n))
- ***Sufficiency part:***
  g(n) = Ω(f(n)) ⇒ f(n) = O(g(n))
  By the definition of Omega (Ω), for some positive constant c ⇒ g(n) ≥ c.f(n) ⇒ f(n) ≤ (1/c).g(n)
  By the definition of Big-Oh(O), f(n) = O(g(n))

Similarly,

f(n) = o(g(n)) if and only if g(n) = ω(f(n))

5. Since these properties hold for asymptotic notations, analogies can be drawn between functions f(n) and g(n) and two real numbers a and b.
   - *g(n) = O(f(n)) is similar to a ≤ b*
   - *g(n) = Ω(f(n)) is similar to a ≥ b*
   - *g(n) = Θ(f(n)) is similar to a = b*
   - *g(n) = o(f(n)) is similar to a < b*
   - *g(n) = ω(f(n)) is similar to a > b*
6. **Observations:**

max(f(n), g(n)) = Θ(f(n) + g(n))

Proof:
Without loss of generality, assume f(n) ≤ g(n), ⇒ max(f(n), g(n)) = g(n)
Consider, g(n) ≤ max(f(n), g(n)) ≤ g(n)
⇒ g(n) ≤ max(f(n), g(n)) ≤ f(n) + g(n)

⇒ g(n)/2 + g(n)/2 ≤ max(f(n), g(n)) ≤ f(n) + g(n)
From what we assumed, we can write
⇒ f(n)/2 + g(n)/2 ≤ max(f(n), g(n)) ≤ f(n) + g(n)
⇒ (f(n) + g(n))/2 ≤ max(f(n), g(n)) ≤ f(n) + g(n)
By the definition of θ, max(f(n), g(n)) = θ(f(n) + g(n))

7. *O(f(n)) + O(g(n)) = O(max(f(n), g(n)))*

Proof:
Without loss of generality, assume f(n) ≤ g(n)
⇒ O(f(n)) + O(g(n)) = c1.f(n) + c2.g(n)
From what we assumed, we can write
O(f(n)) + O(g(n)) ≤ c1.g(n) + c2.g(n)
≤ (c1 + c2) g(n)
≤ c.g(n)
≤ c.max(f(n), g(n))
By the definition of Big-Oh(O),
O(f(n)) + O(g(n)) = O(max(f(n), g(n)))

Refer: https://www.youtube.com/watch?v=bxgTDN9c6rg
Designing of algorithms:

Recursion:
The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily.

Advantages of recursion

1. The code may be easier to write.

2. To solve such problems which are naturally recursive such as tower of Hanoi.
3. Reduce unnecessary calling of function.
4. Extremely useful when applying the same solution.
5. Recursion reduce the length of code.
6. It is very useful in solving the data structure problem.
7. Stacks evolutions and infix, prefix, postfix evaluations etc.

Disadvantages of recursion

1. Recursive functions are generally slower than non-recursive function.
2. It may require a lot of memory space to hold intermediate results on the system stacks.
3. Hard to analyze or understand the code.
4. It is not more efficient in terms of space and time complexity.
5. The computer may run out of memory if the recursive calls are not properly checked.

Divide and Conquer:

it is an algorithmic paradigm. A typical Divide and Conquer algorithm solves a problem using following three steps:

1. **Divide** the problem into a number of sub-problems that are smaller instances of the same problem.
2. **Conquer** the sub-problems by solving them recursively. If the sub-problem sizes are small enough, however, just solve the sub-problems in a straightforward manner.

3. **Combine** the solutions to the sub-problems into the solution for the original problem.

Recurrences

Recurrences go hand in hand with the divide-and-conquer paradigm, because they give us a natural way to characterize the running times of divide-and-conquer algorithms. A recurrence is an equation or inequality that describes a function in terms

There are three methods for solving recurrences—that is, for obtaining asymptotic "O" or "θ" bounds on the solution:

- In the **substitution method**, we guess a bound and then use mathematical induction to prove our guess correct.
- **The recursion-tree method** converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use **techniques for bounding summations to solve the recurrence.**
- **The master method** provides bounds for recurrences of the form

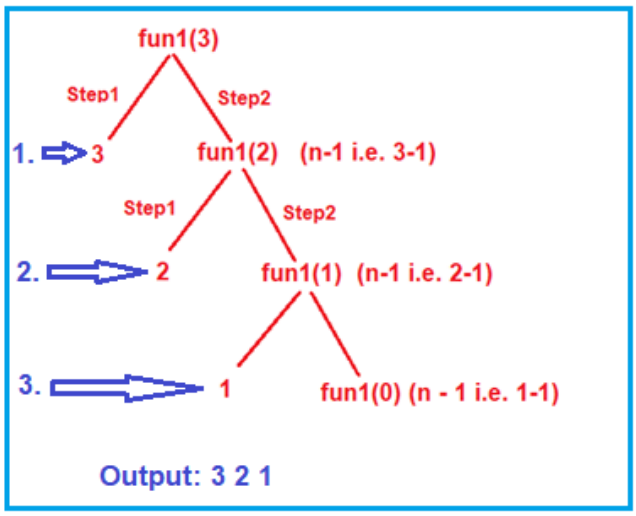
The Examples below will explain how to implement each method.

Example sums for time complexity for recursive functions:
Time Complexity of a Recursive Function:
Note: T(n) is time function.
Here we will be using re-occurrence relation to solve the function time complexity
Case : t(n-1)+1

```
void fun1 (int n){
    if(n>0) {
        printf("%d",n);
```

```
        fun1(n-1);
    }
}
```

Recursion Tree:



NB: 3,2,1 are output on each recursive call.
Consider this function with the recursive call

```
void fun1 (int n){ -- this executes for t(n) time
    if(n>0) { -- this executes 1 time (need not to take)
        printf("%d",n);-- takes 1 unit time
        fun1(n-1); -- takes t(n-1)
    }
}
```

summing up all the comments:
t(n) = t(n-1)+2 (or t(n-1) +1 if you are not considering if)

therefore the recurrence relation looks like this

$$T(n)=\begin{cases} 1 & n=0 \\ T(n-1)+2 & n>0 \end{cases}$$

Induction Method or Successive Substitution method:

We can also solve this using the induction method also called as successive substitution method and we can get the answer. So, let us solve this one. Before solving this, we should know one thing, if we have any constant value there then we should write it as one 1. In our example, the constant value 2 is there, so replace it with 1 as shown below.

So, the recurrence is **T(n)=T(n-1) + 1 ———-[eq.1]**

We can solve this if we know what is T(n-1)
Since,

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

So, we can substitute T(n-2) +1 in place of T(n-1). So, the next equation is
T(n)=T(n-2) + 1 + 1
T(n) =T(n-2) + 2 ———[eq.2]

Let us substitute T(n-3) +1 in that place then this will

be,

T(n)=T(n-3) +1+2

T(n) =T(n-3) +3 ———-[eq.3]

So, we have substituted two times how long we should do this, let us continue it for K times.

T(n)=T(n-k) +k ———[eq.4]

So, go on substituting until it reduces down to a smaller value that is n=0. When we don't know the answer for a bigger expression, then break the bigger one into a smaller one and solve it. The same thing we have done and we don't know how much this is, but we know when n=0 then the answer is directly 1. We have tried to reduce this and by substituting and we got that one.

Now, we see that this n-k actually has become 0. Then assume that **n-k=0**. It means **n=k**. If we substitute that in [eq.4] it gives,

T(n)=T(n-n) +n
=T(0) +n
=1+n

That solves we got the answer **T(n)=1+n**. This can be written as **O(n)**. Earlier directly from their tracing tree we also have seen n+1 was the number of calls and the time taken by this fun1 function depends on the number of calls.

Let us do the exact same thing in forward substitution method

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

substituting values of n =1 ,2,3 and observing the pattern carefully.

T(1)=T(1-1) +1 = T(0)+1 = 1+1

T(2) = T(2-1) + 1 = T(1)+1 = (1+1)+1
T(3) = T(3-1) + 1 = T(2)+1 = (1+1+1)+1
… … … …
T(k) = k+1
put k =n
therefore time complexity eqn = n+1 = O(n)
Case 2: T(n) = T(n-1) +n

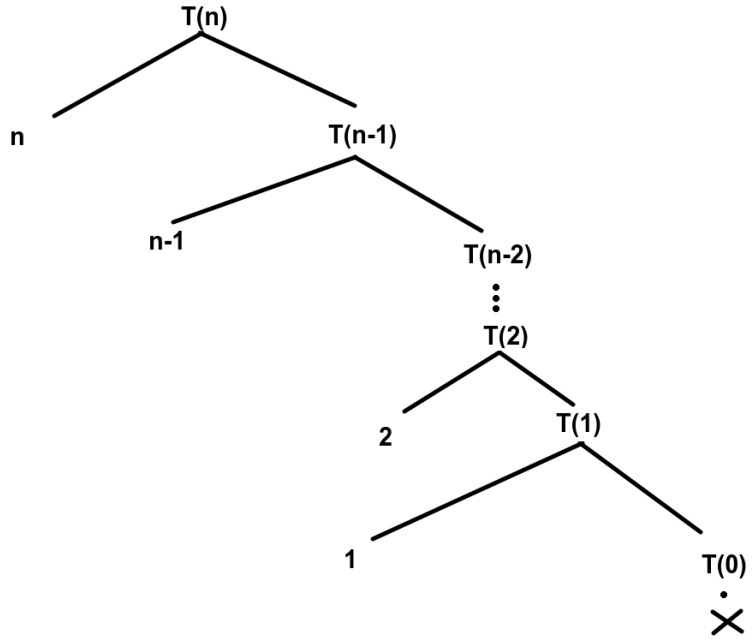
```
void fun1 (int n){
    if(n>0) {
     for(int I =0 ; I < n; ++i)
          printf("%d",n);
         fun1(n-1);
    }
}
```

```
void fun1 (int n){
    if(n>0) { -- takes 1 time
     for(int I =0 ; I < n; ++i)  ---  n + 1
          printf("%d",n); -- n
         fun1(n-1); -- T(n-1)
    }
}
```

therefore: T(n) = T(n-1)+2n+2 = T(n-1) + n

Recursion Tree solution:

Note: n, n-1 …,2,1 is the no of times of execution of for loop inside the recursive function.



Now adding up amount of time the for loop has run we get

For Substitution method you can refer the playlist of Abdul Bari (Algorithms)

$$T(n)=0+1+2+3..n$$
$$T(n)=\frac{(n(n+1))}{2}$$
$$T(n)=\frac{(n^2+n)}{2}$$
$$T(n)=n^2$$

there fore it is $O(n^2)$

Case: T(n)= T(n-1) + log n

```
void fun1 (int n){
    if(n>0) { -- takes 1 time
```

```
     for(int i =0 ; i < n; i*=  2)
          printf("%d",n); -- log 2 n
         fun1(n-1); -- T(n-1) -- T(n-1)
    }
}
```

Therefore : T(n)= T(n-1) + log n

Here : T(1) = 1 and T(0) = 0

Solution by substation:

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

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

= T(n - 3) + log (n - 2) + log (n - 1) + log n

= T(0) + log 1 + log 2 + ... + log (n - 1) + log n

= T(0) + log n! = θ(n log n)

note - using Stirling's approximation, θ(log n!) = θ(n log n). That might help you relate this back to existing complexity classes.

We can just consider n log n for our case. It's not equal, only asymptotically equal.

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

```
void fun1 (int n){
    if(n>0) {
    for(int I =0 ; I < n; ++i)
          printf("%d",n); -- n
```

```
        fun1(n/2);
        fun1(n/2);
    }
}
```

The time function will be T(n) = 2 T(n/2) + n

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

let us solve this. We need to find T(n/2) here

$T(n/2)=2[T(n/2^2)]+n/2$ (eqn. 2)

put (2) in (1) we get

$T(n)=2[2[T(n/2^2)]]+n/2+n=2^2T(n/2^2)¿+2n$

Just try to find $T(n/2^2)$ in the similar manner after proceeding like this

we can see that it follows the below pattern

$2^kT(n/2^K)+kn$

We know that function quits when it is $T(1)$

therefore assuming that $\frac{n}{(2^k)}=1$

now let us find out k value

$n=2k$

$k=\log_2 n$

$put\ k=\log_2 n$ in $2^kT(n/2^K)+kn$ and $n/2^k=1$

$2^{(\log n)}T(1)+(\log n)n$

from the logarithm rule 9 we can simplify this to

$n+(\log n)*n$

which is asymptotically equal to n log n

therefore time complexity is O(n log n)

Refer the following channel (videos) for some examples:

1) Gate Hub
2) Gate Applied course
3) randerson112358

Master Theorem

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

where,
n = size of input
a = number of sub-problems in the recursion
n/b = size of each sub-problem. All sub-problems are assumed to have the same size.
f(n) = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions

Here, a ≥ 1 and b > 1 are constants, and f(n) is an asymptotically positive function.

n asymptotically positive function means that for a sufficiently large value of *n*, we have f(n) > 0.

If a ≥ 1 and b > 1 are constants and f(n) is an asymptotically positive function, then the time complexity of a recursive relation is given by

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

where, T(n) has the following asymptotic bounds:

1. $If\ f(n) = O(n \log_b a - \epsilon), then\ T(n) = \Theta(n \log_b a).$
in other words
$If\ f(n) < O(n \log_b a), then\ T(n) = \Theta(n \log_b a).$
2. $If\ f(n) = \Theta(n log_b a), then\ T(n) = \Theta(n \log_b a * \log n).$
3. $If\ f(n) = \Omega(n \log_b a + \epsilon), then\ T(n) = \Theta(f(n)).$
in other words
$If\ f(n) > O(n \log_b a), then\ T(n) = \Theta(n \log_b a).$

$\epsilon > 0$ is a constant.

each of the above conditions can be interpreted as:

- If the cost of solving the sub-problems at each level increases by a certain factor, the value of f(n) will become polynomially smaller than $n^{\log_b a}$. Thus, the time complexity is oppressed by the cost of the last level ie. $n^{\log_b a}$
- If the cost of solving the sub-problem at each level is nearly equal, then the value of f(n) will be $n^{\log_b a}$. Thus, the time complexity will be f(n) times the total number of levels ie. $n^{\log_b a} * \log n$
- If the cost of solving the sub-problems at each level decreases by a certain factor, the value of f(n) will become polynomially larger than $n^{\log_b a}$. Thus, the time complexity is oppressed by the cost of f(n).

Solved Example of Master Theorem
$T(n) = 3T(n/2) + n^2$
Here,
a = 3
n/b = n/2

$f(n) = n^2$

$\log_b a = \log_2 3 \approx 1.58 < 2$

ie. $f(n) < n^{\log_b a+\epsilon}$, where, ϵ is a constant.

Case 3 implies here.
Thus, $T(n) = f(n) = \theta(n^2)$

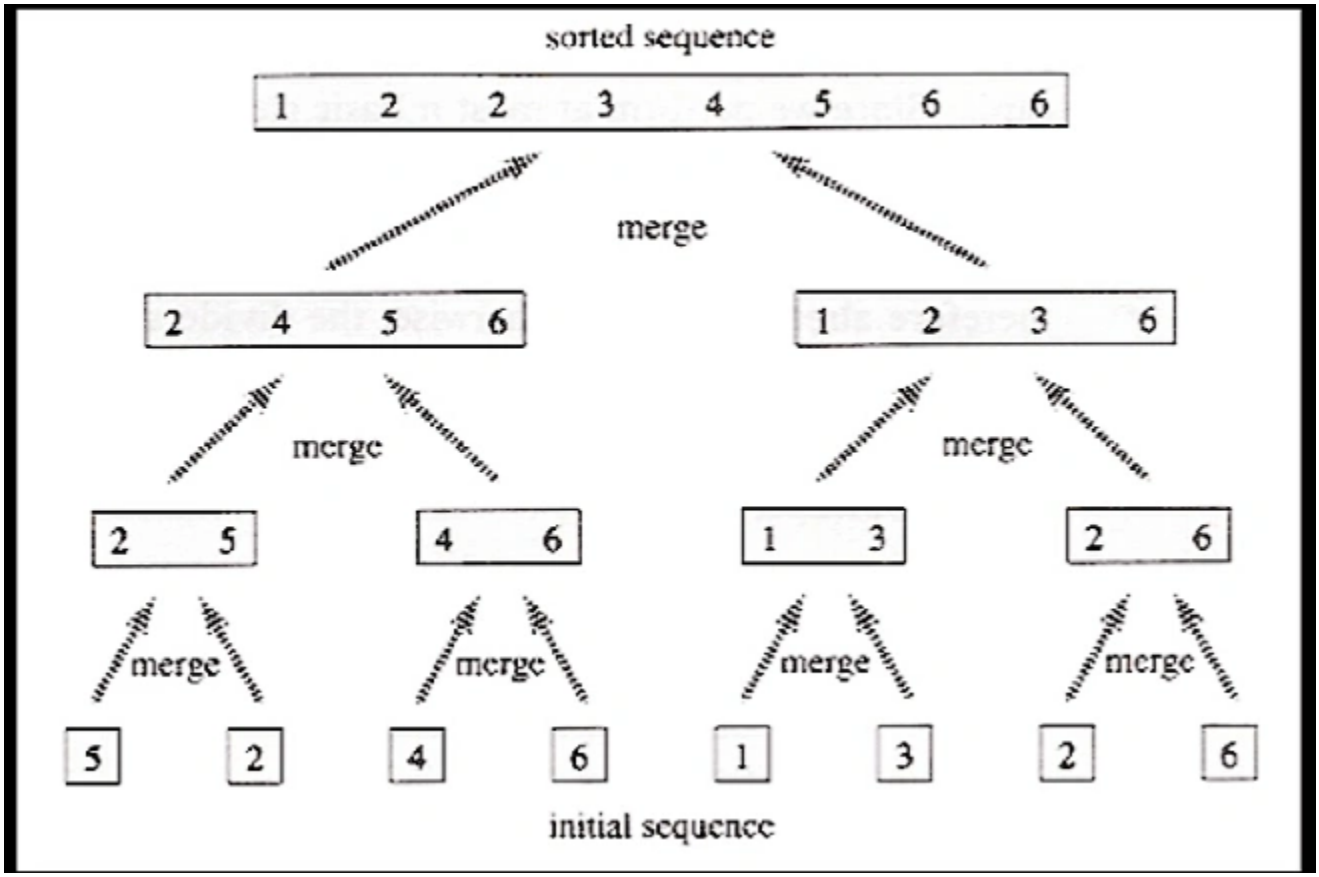
Merge Sort
The merge sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows:

Divide: Divide the n-element sequence to be sorted into two subsequences of n=2 elements each.
Conquer: Sort the two subsequences recursively using merge sort.
Combine: Merge the two sorted subsequences to produce the sorted answer.

Initialization:	Prior to the first iteration of the loop, we have k = p, so that the sub array A [p … k-1] is empty. This empty sub array contains the k – p = 0 smallest elements of L and R, and since i = j = 1, both L [i] and R [j] are the Smallest elements of their arrays that have not been copied back into A.
Maintenance:	To see that each iteration maintains the loop invariant, let us first suppose that L[i] <= R[j]. Then L [i] is the smallest element and copy L [i] into A[k], the sub array A [p… k] will contain the k – p + 1 smallest element. Incrementing k and i reestablishes the loop invariant for the

next iteration. If instead L [i] > R[j], then perform the appropriate action to maintain the loop invariant.

Termination: At termination, k = r + 1. By the loop invariant, the sub array A [p … k- 1], which is A [p … r], contains the k – P = r – p + 1 smallest elements of L [1 … n1 + 1] and R [1 … n2 + 1], in sorted order.



Pseudo code:

Algorithm:

Merge-Sort (numbers[], p, r)

```
if p < r then
   q = ⌊(p + r) / 2⌋
```

```
   Merge-Sort (numbers[], p, q)
// This will take n/2 cycles to complete
    Merge-Sort (numbers[], q + 1, r)
// This will take n/2 cycles to complete
    Merge (numbers[], p, q, r)

end
```

Function: Merge (numbers[], p, q, r)
// This simply runs for n times since if you add the times complexities of for loop it is asymptotically equal to n

$n_1 = q - p + 1$
$n_2 = r - q$
declare $L[1 \ldots n_1 + 1]$ and $R[1 \ldots n2 + 1]$ temporary arrays
for i = 1 to n_1
   L[i] = numbers[p + i - 1]
for j = 1 to n_2
   R[j] = numbers[q+ j]
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
i = 1
j = 1
for k = p to r
   if $L[i] \leq R[j]$
      numbers[k] = L[i]
      i = i + 1
   else
      numbers[k] = R[j]
      j = j + 1

Analysis of merge sort

We assume that we're sorting a total of n elements in the entire array.

Divide: The divide step just computes the middle of the sub array, which takes constant time. Thus, D (n) = (1).

Conquer: We recursively solve two sub problems, each of size n/2, which contributes 2T (n/2) to the running time.

Combine: We have already noted that the MERGE procedure on an n-element sub array takes time (n), so C (n) = (n).

Joining the both time complexities we get
T(n) = 2 T(n/2) + n (Solution can be found in one of examples).

Maximum Sum sub array problem
Note:
refer page 71 in the book named introduction to algorithms for example refer DAA Class notes please. I will add it later
Problem: Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return *its sum*.
Algorithm:

```
1. FIND-MAX-CROSSING-SUBARRAY (A, low, mid, high)
2.   left-sum = -infinity
3.   sum =0
4.   for i = mid down to low
5.     sum + = A[i]
6.     if sum > left-sum
7.       left-sum = sum
8.       max-left = i
```

```
  9.       right-sum = -infinity
  10.      sum = 0
  11.     for j = mid + 1 to high
  12.       sum = sum + A[i]
  13.       if sum > right-sum
  14.         right-sum = sum
  15.         max-right = j
  16.   return (max-left, max-right, left-sum + right-sum)
```

FIND-MAXIMUM-SUB ARRAY (A, low, high):

1. if high == low
2. return(low,high,A[low])
3. else mid = floor((low + high) /2)
4. (left-low,left-high,left-sum)= FIND-MAXIMUM-SUB ARRAY (A, low, mid)
5. (left-low,left-high,left-sum)= FIND-MAXIMUM-SUB ARRAY (A, mid+1,high)
6. (cross-low,cross-high,cross-sum) =**FIND-MAXIMUM-SUB ARRAY (A,low,mid,high)**
7. **if left-sum >= right-sum and left-sum >= cross-sum**
8. **return (left-low, left-high,left-sum)**
9. **else if right-sum >= left-sum and right-sum >= cross-sum return (right-low, right-high,right-sum**
10. **else** (cross-low,cross-high,cross-sum)

Time complexity is same as merge sort.