

01. Algorithms. Time Complexity. Merge Sort.

Algorithms

*Def. **Algorithm:*** Basically a formalized way to solve a problem.

Input Data -> [Algorithm] -> Output Data

Eg. Calculate sum of numbers in an array.

Input: $a[0..n-1]$

Output: $\sum a[i]$

```
S = 0
for i=0..n-1:
    s+=a[i]
print(s)
```

Time Complexity

We can't measure the running time of an algorithm in seconds as different computers will give different speeds, based on the computer's hardware.

*Def. **Computational Model:*** Mathematical abstraction that is used to calculate time and memory complexity.

*Eg. **RAM-Model***

Basically simulates the regular processor in the computer.

Think of memory as one big array. Any element can be accessed in one operation in the RAM model.

```
S = 0 //1 operation
for i=0..n-1: //2 operations per loop. So 2n
    s+=a[i] //3 operations (access, add, assign) per loop. So 3n
print(s) //1 operation
```

$$T(n) = 2 + 5n$$

This is a function which explains how time grows when you increase the size of input.

We take $T(n)$ and remove what is not important.

$T(n) = 5n$, since 2 is not important compared to $5n$

$T(n) = n$, since 5 is not important as it depends on various things and is not a property of the algorithm.

In $T(n) = a + bn$, a and b might vary, but n will remain.

$T(n) = O(n)$

$O(n)$ is the upper bound of the time

If $f(n) = O(g(n))$

$\exists n_0, c \forall n \geq n_0 : f(n) \leq c \cdot g(n)$

Proof. $2+5n = O(n)$

$f(n) = 2+5n$

$g(n) = n$

Consider $n_0=2$ and $c=6$

$2+5n \leq 6n$

$2 \leq n$

So the proof works.

Proof. $n = O(n^2)$

$f(n) = n$

$g(n) = n^2$

Consider $n_0=1$ and $c=1$

$n \leq n^2$

$1 \leq n$

If $f(n) = \Omega(g(n))$

$\exists n_0, c \forall n \geq n_0 : f(n) \geq c \cdot g(n)$

Proof. $2+5n = \Omega(n)$

Consider $n_0=1$ and $c=1$

$2+5n \geq n$

So $T(n) = O(n)$ and $T(n) = \Omega(n)$

If upper and lower bounds are same, $T(n) = \Theta(n)$

How to calculate

eg.

```
for i=0..n-1
  for j=0..n-1
```

$O(n^2)$

eg.

```
for i=0..n-1
    for j=0..i-1 //1+2+3..n-1
```

$O(n^2)$

eg.

```
i=0
while i*i<n
    i++;
```

$O(\sqrt{n})$

eg.

```
i=1
while i<n
    i*=2
```

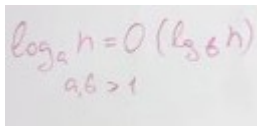
After k iterations, $i=2^k$

$2^k \geq n$

$k \geq \log n$

$O(\log n)$

Note:



$\log_a n = O(\log_b n)$
 $a, b > 1$

Recursive Algorithms

```
def f(n)
    if n=0
        return
    f(n-1)
```

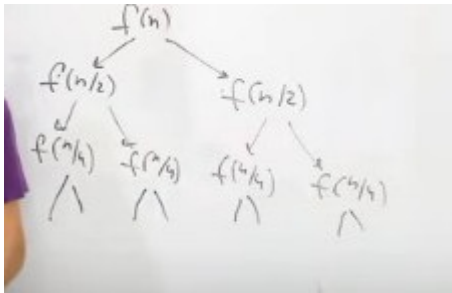
To calculate time complexity, calculate no of times we call the recursive function, and for every call, calculate no of operations.

$f(n) \rightarrow f(n-1) \rightarrow f(n-2) \rightarrow \dots \rightarrow f(0)$

n recursive calls, each of $O(1)$, so $O(n)$

```
def f(n)
    if n=0
        return
    f(n/2)
    f(n/2)
```

We have a tree of recursive calls



To calculate no. of recursive calls, we find the number of nodes.

Height of tree, $H = \log_2(n)$

Total elements = $2^H = 2^{\log_2(n)} = n$

So $O(n)$

```
def f(n)
    if n=0
        return
    f(n/2)
    f(n/2)
    f(n/2)
```

Height of tree, $H = \log_2(n)$

Total elements = $3^H = 3^{\log_2(n)} = n^{\log_2(3)}$

Sorting Algorithms

Input: $a[0..n-1]$

Output: $b[0..n-1]$, same as a , but sorted

Insertion Sort

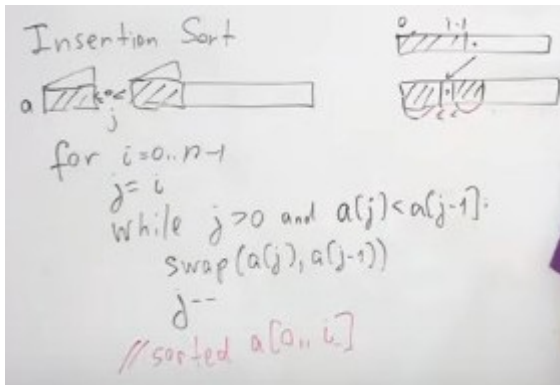
```
for i=0..n-1 //For every element in the array
    j=i //Take the element
```

```

while j>0 and a[j]<a[j-1]: //While index is positive and element is smaller than
its left neighbor (i.e. unsorted)
    swap(a[j], a[j-1]) //Swap element with the left neighbor
    j-- //Decrease the index
//At this point we have sorted a[0..i]

```

Note: Correctedness can be proved using the invariant that after every iteration we have a sorted prefix.



Time complexity:

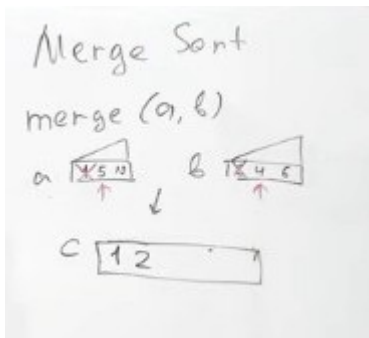
Best case: Array is already sorted. Traverse all elements once. $O(n)$

Worst case: Array sorted in reverse. $O(n^2)$

$\therefore O(n^2)$

Merge Sort

`merge(a,b)` merges two sorted arrays into one sorted array.



`merge(a,b)`

`n = len(a) //length of array a`

`m = len(b) //length of array b`

`i=0, j=0, k=0`

`c = int[n+m] //array of size n+m`

`while i<m or j<m: //while there are elements`

`if j=m or (i<n and a[i]<b[j]) //If 2nd array is empty`

`// or if element of a < element of b`

`c[k++]=a[k++] //move element of a to array c`

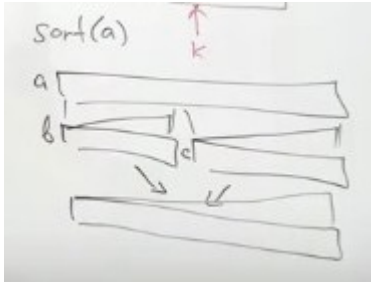
```

//and increase counters
else
    c[k++]=b[j++] //move element of b to array c
    //and increase counters
return c //c is sorted array

```

Time complexity: $O(n+m)$

`sort(a)` sorts the array using Divide and Conquer



```

sort(a):
    if len(a)<2 //If array has less than 2 elements
        return a //Return it
    b=[0..(n/2)-1] //First half
    c=[n/2..n-1] //Second half
    b=sort(b) //Recursively sort first half
    c=sort(c) //Recursively sort second half
    return merge(b,c) //Merge both halves

```

Time complexity: $T(n) = 2T(n/2) + c.n$ since two recursive calls of size $n/2$ + linear time merging

= $O(n \log n)$ (either calculate by Recursion tree or by Master Theorem or by induction)

Proof. (by Induction) $T(n) \leq cn \log n$

$T(n) \leq 2.c.(n/2)\log(n/2) + c.n$

$T(n) \leq c.n.\log(n-1) + cn = c.n.\log n$

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

Master Theorem

Suppose a problem is split into 'b' parts and 'a' recursive calls are made, and some extra operations are made.

$T(n) = aT(n/b) + g(n)$

and

$g(n)$'s complexity is some $c \cdot n^k$

Then

$T(n) = n^k$ if $a < b^k$

$T(n) = (n^k) \log n$ if $a = b^k$

$T(n) = n^{(\log b(a))}$ if $a > b^k$