UNIT-2
DIVIDE & CONQUER

Overview:

- Divide and Conquer
- Master theorem
- Master theorem based analysis for
  - Binary Search
  - Merge Sort
  - Quick Sort

Divide and Conquer

Basic Idea:

1. Decompose problems into sub instances.
2. Solve sub instances successively and independently.
3. Combine the sub solutions to obtain the solution to the original problem.

In order to look into the efficiency of the Divide and Conquer lets look into the

Multiplication of two n-digit Numbers

Traditional Multiplication:
Say we are multiplying 382 with 695(n=3)

```
   382
 * 695
  ____
   ___
  ___
___
_____
- - - - -
```

Essentially, we are multiplying 1 digit with n other digits and then adding the n numbers, which can give us a solution of at most 2n digits.

There are n additions, each of O(n) time at most, which gives us the running time of the algorithm as $O(n^2)$

*!!*Using Divide and Conquer to multiply n-digit numbers

We will write the two n-digit numbers as follows:

$$(10^{n/2}X + Y)\,(10^{n/2}W + Z) = 10^{n}XW + (XZ + YW)\,10^{n/2} + YZ \qquad \text{---(1)}$$

That is we are converting the multiplication of two n-digit numbers into multiplication of four n/2 digit numbers, plus some extra work involved in additions. We are recursively calling multiplication and performing some additions in every recursion.

Let T (n) be the running time of multiplying two n-digit numbers.

Then in our case,
$T(n) = 4T(n/2) + O(n)$

- Four multiplications of n/2 digit numbers
- Addition is going to be between numbers that have atmost 2n digits. Thus addition can be O (n).

Recursively substituting the value of T (n):
$T(n) = 4[4T(n/4) + O(n/2)] + O(n)$
$=16\,T(n/4) + 4O(n/2) + O(n)$
-
-
-
$=C\,T(1) + - - - -$

Master's Theorem

Let T(n) be the running time of an algorithm with an input size of n;
Suppose we can run the algorithm in such a way that we make 'a' recursive calls every time with an input size of 'n/b' and do some extra work in every recursion (additions and subtractions).
Such that T (n) can be represented as:

$$T(n) = a\,T(n/b) + O(n^k),$$

Then,
If $\log_b a > k$, $T(n) = O(n^{\log_b a})$ (recursive calls dominates)
If $\log_b a = k$, $T(n) = O(n^k \log n)$ (almost equal work in rec. calls and in extra work)
If $\log_b a < k$, $T(n) = O(n^k)$ (Extra work dominates)

In our multiplication problem:
$T(n) = 4T(n/2) + O(n)$

A=4, b=2
$\text{Log}_2 4 = 2$, k=1

Since algorithm is dominated by recursive calls and the running time is $O(n^2)$.

But this is as good as our traditional multiplication algorithm. Since we now know that multiplications dominate the running time, if we can reduce the number of multiplications to three, which can bring down our T(n) by 25%.

To calculate (1), we just need the following 3 multiplications separately:
1. (X+Y)(W+Z) 2 additions and one multiplication
2.XW
3.YZ

Then we can calculate
XZ+YW=(X+Y)(W+Z)-XW-YZ
Thus we use three multiplications at the expense of some extra additions and subtractions, which run in constant time(each of O(n) time)

Thus,

$$T(n)=3T(n/2) + O(n)$$

Applying Master's theorem,
A=3,b=2,k=1
Thus, $T(n)=O(n^{\log_2 3})$

Since $\log_2 3 \sim 1.5$,
We have reduced the total number of recursive calls in our program. For very large n, it will work well but in actual implementation, we hardly code to gain advantage out if this feature.

Binary Search

Goal: Searching for nth value in a sorted list of length n.
(Divide the list into two and recursively search in the individual lists half the size)

Again,
Let T(n) be the running time of the algorithm. Then,

$$T(n)=T(n/2) + O(1)$$

In O(1) time we divide the list into two halves(n/2) that run in T(n/2) time.

Using Master's theorem,
A=1,b=2
$Log_2 1=0$
K=0;
So,
T(n)=O(log n)

Merge Sort
Goal: Splitting the element list into 2 lists, sort them and merge them.
T(n)=2T(n/2) + O(n)

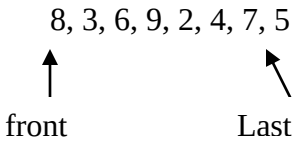
Here, the hidden constant is greater than the hiddent constant in the merge sort because while dividing the lists into two different arrays and then sorting them, we are allocating extra space and subsequently, copying into the original array.

Using Master's theorem,
A=2,b=2,k=1
$Log_2 2=1$
So, T(n)=O(n log n)

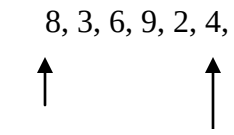
Quicksort

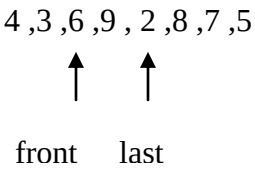
Goal: Pick one element as the partition element. Put all the elements smaller than it, to the left of it and all elements greater than it, to the right of it. On the lists left and right of the partition element recursively call Qsort.

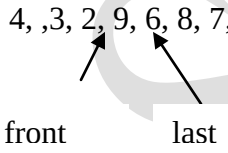
Say the list is: 8,3,6,9,2,4,7,5
Partition element:5

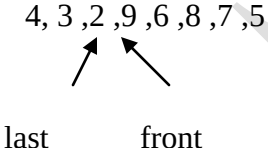


8 in the worng place, 7 fine.









Now swap front with 5 and we have 5 in place.
4,3,2,5,6,8,7,9

Thus the only extra space utilized here is the temporary variable used for swapping.

In te worst case, we might end up choosing a partition element which is the first element in our list.
In that case $T(n)=O(n^2)$

To make sure this rarely happens:

1. Pick a random partition element.
2. Probablity of picking a good partition element is as low as the probability of picking a bad one. So, they will even out.

There are n possible partition elements

Element	**Split**	**Prob(element)**
1	0,n-1	1/n
2	1,n-2	1/n
3		1/n
N	n-1,0	1/n

Now,

$T(n) = 1/n [T(0) + T(n-1) + O(n)] +$
$1/n [T(1) + T(n-2) + O(n)] +$
$1/n [T(2) + T(n-3) + O(n)] +$
……
……
……
$1/n [T (n-1) + T (0) + O (n)]$

$$n * T[n] = \{2 \sum_{k=0}^{n-1} T(k)\} + O(n^2) \rightarrow \mathbf{A}$$

Substitute n = n-1,

$$(n-1) * T[(n-1)] = \{2 \sum_{k=0}^{n-1} T(k)\} + O((n-1)^2) \rightarrow \mathbf{B}$$

*Subtract **A** from **B***

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

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

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

Divide by (n+1)

$T(n)/(n+1) = [T(n-1)]/n + O(1/n) \rightarrow \mathbf{C}$

Let,

$S(n) = T(n)/(n+1) \rightarrow \mathbf{D}$

$S(n) = S(n-1) + O(1/n)$

This can be written as a sum,

$= S(n-2) + O(1/n-1) + O(1/n)$

$= S(n-3) + O(1/n-2) + O(1/n-1) + O(1/n)$

$S(n) = O(\sum_{k=1}^{n} 1/k)$

$= O(H_n)$

$S(n) = T(n)/(n+1) = O(\ln n) \rightarrow \mathbf{E}$

*Substitute **D** in **C***

$T(n) = S(n) \cdot (n+1)$

use **E**,

$T(n) = O(\ln n) \cdot (n+1)$

$T(n) = O(n \lg n)$

Strassen's algorithm for Matrix multiplication:

The standard method of matrix multiplication of two n x n matrices takes

$T(n) = O(n^3)$.

The following is a simple algorithm to implement n x n matrix multiplication:

```
Result = 0;          ( initialize the array of Result )
for i = 1 to n
  for j = 1 to n
    for k = 1 to n
        Result [i, j] += A[i, k] * B[k, j];
```

Stassen's algorithm is a Divide-and-Conquer algorithm that beat the bound. The usual multiplication of two n x n matrices takes

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

8 n/2 * n/2 matrix multiples plus
4 n/2 * n/2 matrix additions

$T(n) = 8T(n/2) + O(n^2)$
Plug in a = 8, b = 2, k = 2 → $\log_b{}^a = 3$ → $T(n) = O(n^3)$

Strassen showed how two matrices can be multiplied using only 7 multiplications and 24 additions:

$m_1 = (a_{21} + a_{22} - a_{11}) * (b_{22} - b_{12} + b_{11})$
$m_2 = a_{11} * b_{11}$
$m_3 = a_{12} * b_{21}$
$m_4 = (a_{11} - a_{21}) * (b_{22} - b_{12})$
$m_5 = (a_{21} + a_{22}) * (b_{12} - b_{11})$
$m_6 = (a_{12} - a_{21} + a_{11} - a_{22}) * b_{22}$
$m_7 = a_{22} * (b_{11} + b_{22} - b_{12} - b_{21})$

$T(n) = 7T(n/2) + \underline{O(n^2)}$
↓
6 times as much as in the other algorithm