LECTURE NOTES

ON

DESIGN AND ANALYSIS OF ALGORITHMS

B. Tech. 6th Semester

Computer Science & Engineering

and

Information Technology

Prepared by

Mr. S.K. Sathua – Module I
Dr. M.R. Kabat – Module II
Dr. R. Mohanty – Module III

VEER SURENDRA SAI UNIVERSITY OF TECHNOLOGY, BURLA

SAMBALPUR, ODISHA, INDIA – 768018

CONTENTS

MODULE - I

- Lecture 1 - Introduction to Design and analysis of algorithms
- Lecture 2 - Growth of Functions (Asymptotic notations)
- Lecture 3 - Recurrences, Solution of Recurrences by substitution
- Lecture 4 - Recursion tree method
- Lecture 5 - Master Method
- Lecture 6 - Design and analysis of Divide and Conquer Algorithms
- Lecture 7 - Worst case analysis of merge sort, quick sort and binary search
- Lecture 8 - Heaps and Heap sort
- Lecture 9 - Priority Queue
- Lecture 10 - Lower Bounds for Sorting

Lecture 1 - Introduction to Design and analysis of algorithms

What is an algorithm?

Algorithm is a *set of steps to complete a task.*

For example,

Task: to make a cup of tea.

Algorithm:

- add water and milk to the kettle,
- boilit, add tea leaves,
- Add sugar, and then serve it in cup.

What is *Computer algorithm*?

''a set of steps to accomplish or complete a task that is described precisely enough that a computer can run it''.

Described precisely: very difficult for a machine to know how much water, milk to be added etc. in the above tea making algorithm.

These algorithmsrun on computers or computational devices.Forexample, GPS in our smartphones, Google hangouts.

GPS uses *shortest path algorithm.* Online shopping uses cryptography which uses RSA algorithm.

Characteristics of an algorithm:-

- Must take an input.
- Must give some output(yes/no,valueetc.)
- Definiteness –each instruction is clear and unambiguous.
- Finiteness –algorithm terminates after a finite number of steps.
- Effectiveness –every instruction must be basic i.e. simple instruction.

Expectation from an algorithm

- Correctness:-
  - Correct: Algorithms must produce correct result.
  - Produce an incorrect answer:Even if it fails to give correct results all the time still there is a control on how often it gives wrong result. Eg.Rabin-Miller PrimalityTest (Used in RSA algorithm): It doesn’t give correct answer all the time.1 out of 2^{50} times it gives incorrect result.
  - Approximation algorithm: Exact solution is not found, but near optimal solution can be found out. (Applied to optimization problem.)
- Less resource usage:

  Algorithms should use less resources (time and space).

Resource usage:

Here, the time is considered to be the primary measure of efficiency .We are also concerned with how much the respective algorithm involves the computer memory.But mostly time is the resource that is dealt with. And the actual running time depends on a variety of backgrounds: like the speed of the Computer, the language in which the algorithm is implemented, the compiler/interpreter, skill of the programmers etc.

So, mainly the resource usage can be divided into: 1.Memory (space) 2.Time

<u>Time taken by an algorithm?</u>

- performance measurement or Apostoriori Analysis: Implementing the algorithm in a machine and then calculating the time taken by the system to execute the program successfully.
- Performance Evaluation or Apriori Analysis. Before implementing the algorithm in a system. This is done as follows

1. How long the algorithm takes :-will be represented as a function of the size of the input.

f(n)→how long it takes if ‘n’ is the size of input.

2. How fast the function that characterizes the running time grows with the input size.
   “Rate of growth of running time”.
   The algorithm with less rate of growth of running time is considered better.

How algorithm is a technology ?

Algorithms are just like a technology. We all use latest and greatest processors but we need to run implementations of good algorithms on that computer in order to properly take benefits of our money that we spent to have the latest processor. Let’s make this example more concrete by pitting a faster computer(computer A) running a sorting algorithm whose running time on n values grows like n^2 against a slower computer (computer B) running asorting algorithm whose running time grows like n lg n. They eachmust sort an array of 10 million numbers. Suppose that computer A executes 10 billion instructions per second (faster than anysingle sequential computer at the time of this writing) and computer B executes only 10 million instructions per second, so that computer A is1000 times faster than computer B in raw computing power. To makethe difference even more dramatic, suppose that the world’s craftiestprogrammer codes in machine language for computer A, and the resulting code requires $2n^2$ instructions to sort n numbers. Suppose furtherthat just an average programmer writes for computer B, using a high-level language with an inefficient compiler, with the resulting code taking 50n lg n instructions.

<u>Computer A (Faster)</u>	<u>Computer B(Slower)</u>
Running time grows like n^2.	Grows innlogn.
10 billion instructions per sec.	10million instruction per sec
$2n^2$ instruction.	50 nlogn instruction.

Time taken=$\frac{2\times(10^7)^2}{10^{10}} = 20{,}000$ $\qquad$ $\frac{50\times10^7 \times\log 10^7}{10^7} \approx 1163$

It is more than 5.5hrs $\qquad$ it is under 20 mins.

So choosing a good algorithm (algorithm with slower rate of growth) as used by computer B affects a lot.

Lecture 2 - Growth of Functions (Asymptotic notations)

Before going for growth of functions and asymptotic notation let us see how to analyse an algorithm.

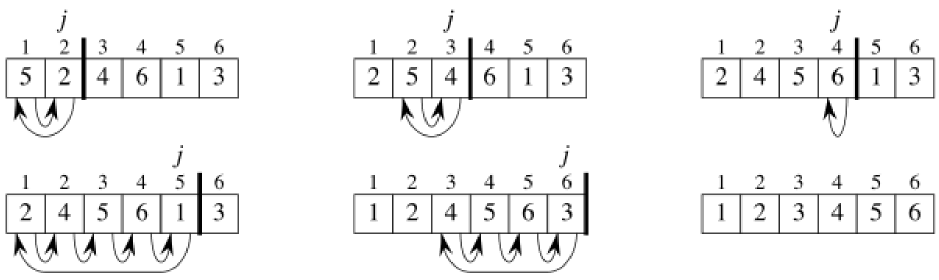
<u>How to analyse an Algorithm</u>

Let us form an algorithm for Insertion sort (which sort a sequence of numbers).The pseudo code for the algorithm is give below.

<u>Pseudo code:</u>

```
for j=2 to A length -------------------------------------------------- $C_1$
key=A[j]-----------------------------------------------------------------$C_2$
//Insert A[j] into sorted Array A[1.....j-1]------------------------$C_3$
i=j-1-----------------------------------------------------------------------$C_4$
while i>0 & A[j]>key--------------------------------------------------$C_5$
A[i+1]=A[i]--------------------------------------------------------------$C_6$
i=i-1-----------------------------------------------------------------------$C_7$
A[i+1]=key---------------------------------------------------------------$C_8$
```

Example:



Let C_i be the cost of i^{th} line. Since comment lines will not incur any cost $C_3=0$.

Cost	No. Of times Executed
C_1	n
C_2	$n-1$
$C_3=0$	$n-1$
C_4	$n-1$
C_5	$\sum_{j=2}^{n-1} t_j$
C_6	$\sum_{j=2}^{n}(t_j - 1)$
C_7	$\sum_{j=2}^{n}(t_j - 1)$
C_8	$n-1$

Running time of the algorithm is:

$$T(n)=C_1n+C_2(n-1)+0(n-1)+C_4(n-1)+C_5\left(\sum_{j=2}^{n-1} t_j\right)+C_6\left(\sum_{j=2}^{n} t_j - 1\right)+C_7\left(\sum_{j=2}^{n} t_j - 1\right)+ C_8(n-1)$$

<u>Best case:</u>

It occurs when Array is sorted.

All $t_j$values are 1.

$T(n)=C_1n+C_2(n-1)+0\ (n-1)+C_4(n-1)+C_5(\sum_{j=2}^{n-1} 1)+C_6(\sum_{j=2}^{n} 0)+C_7(\sum_{j=2}^{n} 0)+ C_8(n-1)$

$=C_1n+C_2\ (n-1)\ +0\ (n-1)\ +C_4\ (n-1)\ +C_5(n-1)+ C_8\ (n-1)$

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

- Which is of the forman+b.
- Linear function of n.So, linear growth.

Worst case:

It occurs when Array is reverse sorted, and $t_j = j$

$T(n)=C_1n+C_2(n-1)+0\ (n-1)+C_4(n-1)+C_5(\sum_{j=2}^{n-1} j)+C_6(\sum_{j=2}^{n} j-1)+C_7(\sum_{j=2}^{n} j-1)+ C_8(n-1)$

$=C_1n+C_2(n-1)+C_4(n-1)+C_5(\frac{n(n-1)}{2}-1)+C_6(\sum_{j=2}^{n}\frac{n(n-1)}{2})+C_7(\sum_{j=2}^{n}\frac{n(n-1)}{2})+ C_8(n-1)$

which is of the form an^2+bn+c

Quadratic function. So in worst case insertion set grows in n^2.

Why we concentrate on worst-case running time?

- The worst-case running time gives a guaranteed upper bound on the runningtime for any input.
- For some algorithms, the worst case occurs often. For example, when searching, the worst case often occurs when the item being searched for is not present, and searches for absent items may be frequent.
- Why not analyze the average case? Because it’s often about as bad as the worst case.

Order of growth:

It is described by the highest degree term of the formula for running time. (Drop lower-order terms. Ignore the constant coefficient in the leading term.)

Example: We found out that for insertion sort the worst-case running time is of the form $an^2 + bn + c$.

Drop lower-order terms. What remains is an^2.Ignore constant coefficient. It results in n^2.But we cannot say that the worst-case running time T(n) equals n^2 .Rather It grows like n^2 . But it doesn't equal n^2.We say that the running time is $\Theta(n^2)$ to capture the notion that the order of growth is n^2.

We usually consider one algorithm to be more efficient than another if its worst-case running time has a smaller order of growth.

Asymptotic notation

- It is a way to describe the characteristics of a function in the limit.
- It describes the rate of growth of functions.
- Focus on what's important by abstracting away low-order terms and constant factors.
- It is a way to compare "sizes" of functions:

  $O \approx \leq$

  $\Omega \approx \geq$

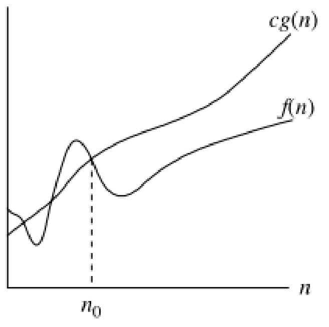
  $\Theta \approx =$

  $o \approx <$

  $\omega \approx >$

O-notation

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



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

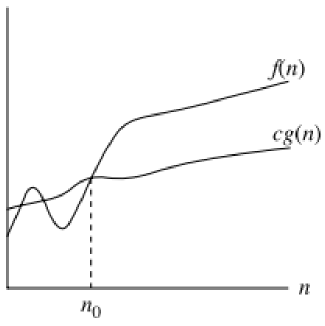
Example: $2n^2 = O(n^3)$, with $c = 1$ and $n_0 = 2$.

Examples of functions in $O(n^2)$:

n^2
$n^2 + n$
$n^2 + 1000n$
$1000n^2 + 1000n$
Also,
n
$n/1000$
$n^{1.99999}$
$n^2/\lg\lg\lg n$

Ω-notation

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



$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

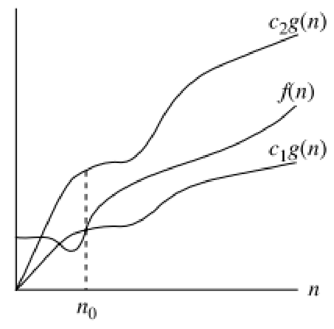
Example: $\sqrt{n} = \Omega(\lg n)$, with $c = 1$ and $n_0 = 16$.

Examples of functions in $\Omega(n^2)$:

n^2
$n^2 + n$
$n^2 - n$
$1000n^2 + 1000n$
$1000n^2 - 1000n$
Also,
n^3
$n^{2.00001}$
$n^2 \lg\lg\lg n$
2^{2^n}

Θ-notation

$\Theta(g(n)) = \{f(n) :$ there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$.



$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Example: $n^2/2 - 2n = \Theta(n^2)$, with $c_1 = 1/4$, $c_2 = 1/2$, and $n_0 = 8$.

***o*-notation**

$o(g(n)) = \{f(n) :$ for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le f(n) < cg(n)$ for all $n \ge n_0\}$.

Another view, probably easier to use: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

$n^{1.9999} = o(n^2)$
$n^2/\lg n = o(n^2)$
$n^2 \ne o(n^2)$ (just like $2 \not< 2$)
$n^2/1000 \ne o(n^2)$

***ω*-notation**

$\omega(g(n)) = \{f(n) :$ for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \le cg(n) < f(n)$ for all $n \ge n_0\}$.

Another view, again, probably easier to use: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

$n^{2.0001} = \omega(n^2)$
$n^2 \lg n = \omega(n^2)$
$n^2 \ne \omega(n^2)$

Lecture 3-5: Recurrences, Solution of Recurrences by substitution, Recursion Tree and Master Method

Recursion is a particularly powerful kind of reduction, which can be described loosely as follows:

• If the given instance of the problem is small or simple enough, just solve it.

• Otherwise, reduce the problem to one or more simpler instances of the same problem.

Recursion is generally expressed in terms of recurrences. In other words, when an algorithm calls to itself, we can often describe its running time by a **recurrence equation** which describes the overall running time of a problem of size n in terms of the running time on smaller inputs.

E.g.the worst case running time T(n) of the merge sort procedure by recurrence can be expressed as

$$T(n) = \begin{cases} \Theta(1) \ ; & \text{if } n=1 \\ 2T(n/2) + \Theta(n) & ;\text{if } n>1 \end{cases}$$

whose solution can be found as $T(n)=\Theta(n\log n)$

There are various techniques to solve recurrences.

1. **SUBSTITUTION METHOD**:

The substitution method comprises of 3 steps

i. Guess the form of the solution
ii. Verify by induction
iii. Solve for constants

We substitute the guessed solution for the function when applying the inductive hypothesis to smaller values. Hence the name “substitution method”. This method is powerful, but we must be able to guess the form of the answer in order to apply it.

e.g.recurrence equation: $T(n)=4T(n/2)+n$

step 1: guess the form of solution

$T(n)=4T(n/2)$

- ⇨ $F(n)=4f(n/2)$
- ⇨ $F(2n)=4f(n)$
- ⇨ $F(n)=n^2$

  So, $T(n)$ is order of n^2

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

Step 2: verify the induction

Assume $T(k)<=ck^3$

$T(n)=4T(n/2)+n$

$<=4c(n/2)^3 +n$

$<=cn^3/2+n$

$<=cn^3-(cn^3/2-n)$

$T(n)<=cn^3$ as $(cn^3/2 –n)$ is always positive

So what we assumed was true.

- ⇨ $T(n)=O(n^3)$

Step 3: solve for constants

$Cn^3/2-n>=0$

- ⇨ $n>=1$
- ⇨ $c>=2$

Now suppose we guess that $T(n)=O(n^2)$ which is tight upper bound

Assume,$T(k)<=ck^2$

so,we should prove that $T(n)<=cn^2$

$T(n)=4T(n/2)+n$

- ⇨ $4c(n/2)^2+n$
- ⇨ cn^2+n

So,T(n) will never be less than cn^2. But if we will take the assumption of $T(k)=c_1 k^2-c_2k$, then we can find that $T(n) = O(n^2)$

2. BY ITERATIVE METHOD:

e.g. $T(n)=2T(n/2)+n$

$\Rightarrow 2[2T(n/4) + n/2]+n$

$\Rightarrow 2^2T(n/4)+n+n$

$\Rightarrow 2^2[2T(n/8)+ n/4]+2n$

$\Rightarrow 2^3T(n/2^{3)} +3n$

After k iterations ,$T(n)=2^kT(n/2^k)+kn$--------------(1)

Sub problem size is 1 after $n/2^k=1 \Rightarrow k=\log n$

So,afterlogn iterations ,the sub-problem size will be 1.

So,when k=logn is put in equation 1

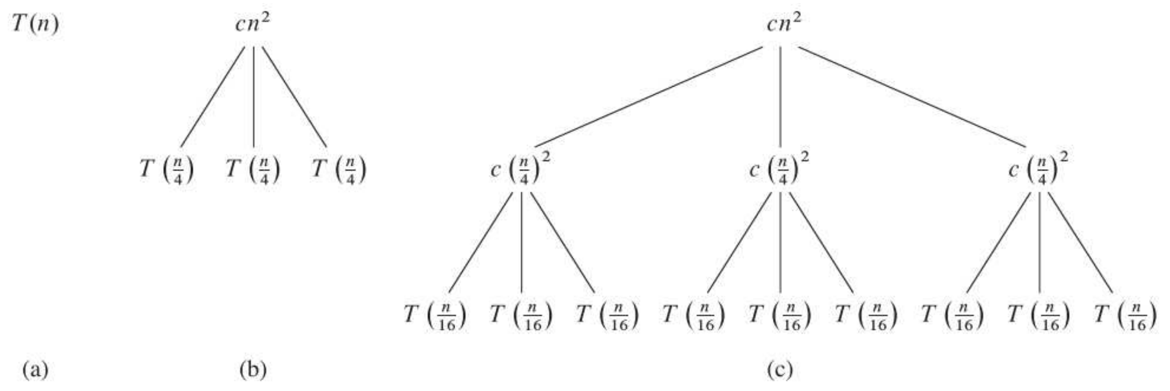
$T(n)=nT(1)+n\log n$

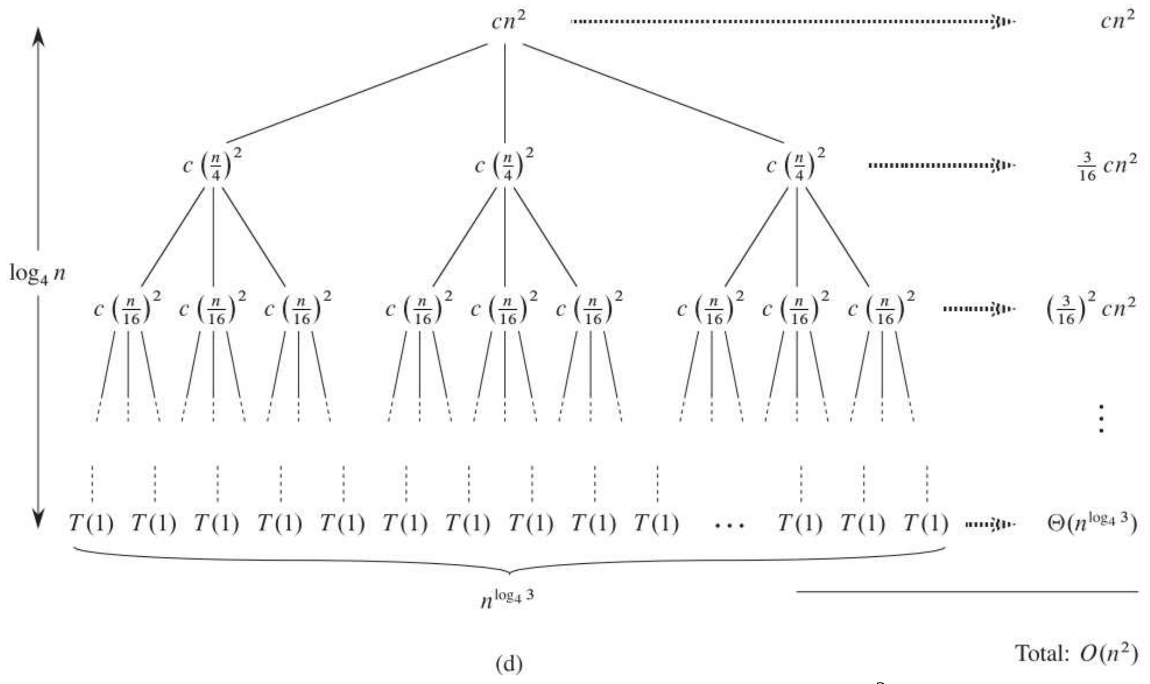
- ⇨ $nc+n\log n$ (say $c=T(1)$)
- ⇨ $O(n\log n)$

3.BY RECURSSION TREE METHOD:

In a recursion tree ,each node represents the cost of a single sub-problem somewhere in the set of recursive problems invocations .we sum the cost within each level of the tree to obtain a set of per level cost,and then we sum all the per level cost to determine the total cost of all levels of recursion .

Constructing a recursion tree for the recurrence $T(n)=3T(n/4)+cn^2$


$T(n)$
cn^2
$T\left(\frac{n}{4}\right)$
$c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$
(a)
(b)
(c)



Constructing a recursion tree for the recurrence T (n)= 3T (n=4) + cn^2.. Part (a) shows T (n), which progressively expands in (b)–(d) to form the recursion tree. The fully expanded tree in part (d) has height $\log_4 n$ (it has $\log_4 n + 1$ levels).

Sub problem size at depth i $=n/4^i$

Sub problem size is 1 when $n/4^i=1 \Rightarrow i=\log_4 n$

So, no. of levels $=1+ \log_4 n$

Cost of each level = (no. of nodes) x (cost of each node)

No. Of nodes at depth i=3^i

Cost of each node at depth i=$c\,(n/4^i)^2$

Cost of each level at depth i=$3^i\,c\,(n/4^i)^2 = (3/16)^i cn^2$

$T(n)= {}_{i=0}\sum^{\log_4 n}\; cn^2(3/16)^i$

$T(n)= {}_{i=0}\sum^{\log_4 n\,-1}\; cn^2(3/16)^i$ + cost of last level

Cost of nodes in last level =$3^i T(1)$

- ⇨ $c3^{\log_4 n}$ (at last level i=$\log_4 n$)
- ⇨ $cn^{\log_4 3}$

$$T(n)=\sum_{i=0}^{\log_4 n\,-1} cn^2(3/16)^i \;+\; c\,n^{\log_4 3}$$

$$<= \; cn^2\sum_{i=0}^{\infty} (3/16)^i + cn^{\log_4 3}$$

- ⇨ $<= cn^2*(16/13)+ cn^{\log_4 3}$ => $T(n)=O(n^2)$

<u>4.BY MASTER METHOD:</u>

The master method solves recurrences of the form

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

where $a>=1$ and $b>1$ are constants and f(n) is a asymptotically positive function .

To use the master method, we have to remember 3 cases:

1. If $f(n)=O(n^{\log_b a - \varepsilon})$ for some constants $\varepsilon >0$,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 $f(n)=\Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon>0$,and if $a*f(n/b)<=c*f(n)$ for some constant $c<1$ and all sufficiently large n,then $T(n)=\Theta(f(n))$

e.g. $T(n)=2T(n/2)+n\log n$

ans: $a=2$ $b=2$

$f(n)=n\log n$

using 2nd formula

$f(n)=\Theta(n^{\log_2 2}\log^k n)$

$\Rightarrow \Theta(n^1 \log^k n)=n\log n$ $\Rightarrow K=1$

$T(n)=\Theta(n^{\log_2 2} \log^1 n)$

$\Rightarrow \Theta(n\log^2 n)$

Lecture 6 - Design and analysis of Divide and Conquer Algorithms

DIVIDE AND CONQUER ALGORITHM

- In this approach ,we solve a problem recursively by applying 3 steps

1. **DIVIDE-**break the problem into several sub problems of smaller size.
2. **CONQUER**-solve the problem recursively.
3. **COMBINE-**combine these solutions to create a solution to the original problem.

CONTROL ABSTRACTION FOR DIVIDE AND CONQUER ALGORITHM

```
Algorithm D and C (P)
{
    if small(P)
        then return S(P)
    else
        {   divide P into smaller instances $P_1$ ,$P_2$ .....$P_k$
        Apply D and C to each sub problem
            Return combine (D and C($P_1$)+ D and C($P_2$)+.......+D and C($P_k$))
    }
```

}

Let a recurrence relation is expressed as T(n)=

$\Theta(1)$, if $n<=C$

$aT(n/b) + D(n)+ C(n)$,otherwise

then n=input size a=no. Of sub-problemsn/b= input size of the sub-problems

Lecture 7: Worst case analysis of merge sort, quick sort

Merge sort

It is one of the well-known divide-and-conquer algorithm. This is a simple and very efficient algorithm for sorting a list of numbers.

We are given a sequence of n numberswhich we will assume is stored in an array A [1...n]. Theobjective is to output a permutation of this sequence, sorted in increasing order. This is normally done by permuting the elements within the array A.

How can we apply divide-and-conquer to sorting? Here are the major elements of the Merge Sort algorithm.

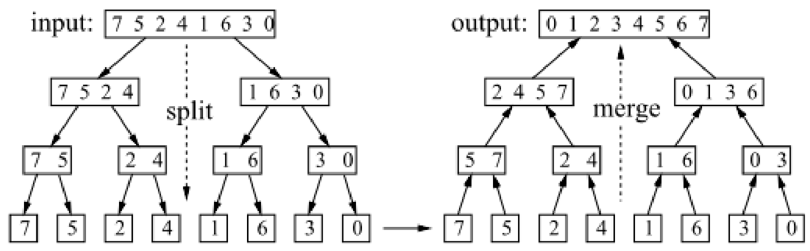
Divide: Split A down the middle into two sub-sequences, each of size roughly n/2 .

Conquer: Sort each subsequence (by calling MergeSort recursively on each).

Combine: Merge the two sorted sub-sequences into a single sorted list.

The dividing process ends when we have split the sub-sequences down to a single item. A sequence of length one is trivially sorted. The key operation where all the work is done is in the combine stage,which merges together two sorted lists into a single sorted list. It turns out that the merging process is quite easy to implement.

The following figure gives a high-level view of the algorithm. The “divide” phase is shown on the left. It works top-down splitting up the list into smaller sublists. The “conquer and combine” phases areshown on the right. They work bottom-up, merging sorted lists together into larger sorted lists.



Merge Sort

Designing the Merge Sort algorithm top-down. We’ll assume that the procedure thatmerges two sortedlist is available to us. We’ll implement it later. Because the algorithm is called recursively on sublists,in addition to passing in the array itself, we will pass in two indices, which indicate the first and lastindices of the subarray that we are to sort. The call MergeSort(A, p, r) will sort the sub-arrayA [p..r] and return the sorted result in the same subarray.

Here is the overview. If r = p, then this means that there is only one element to sort, and we may returnimmediately. Otherwise (if p < r) there are at least two elements, and we will invoke the divide-and-conquer. We find the index q, midway between p and r, namely q = (p + r) / 2 (rounded down to thenearest integer). Then we split the array into subarrays A [p..q] and A [q + 1 ..r] . Call Merge Sort recursively to sort each subarray. Finally, we invoke a procedure (which we have yet to write) whichmerges these two subarrays into a single sorted array.

```
MergeSort(array A, int p, int r) {
        if      (p < r) {                                   // we have at least 2 items
        q = (p + r)/2
        MergeSort(A, p, q)                          // sort A[p..q]
        MergeSort(A, q+1, r)                        // sort A[q+1..r]
```