Algorithm Design Techniques

- Divide and Conquer
- Greedy Algorithm Design
- Dynamic Programming

i j

2 4 5 7 0 1 3 6

0	1	2	3	4	5	6	7

B

k=p

Substitution Method:

$T(n) <= 2^1 T(n/2^1) + cn$

$T(n/2) = 2T\left(\frac{n}{2^2}\right) + c * (\frac{n}{2})$

$T(n) <= 2[2T\left(\frac{n}{2^2}\right) + c * (\frac{n}{2})] + cn$

$= 2^2 T\left(\frac{n}{2^2}\right) + cn] + cn = 2^2 T\left(\frac{n}{2^2}\right) + 2 \ cn$

$=2^2[2T\left(\frac{n}{2^3}\right) + c(\frac{n}{4})] + 2cn$

$=2^3 T\left(\frac{n}{2^3}\right) + 3cn$

At step k

$=2^k T\left(\frac{n}{2^k}\right) + kcn = 2^{log_2 n} T(1) + cnlog_2 n$ [T(1)=1]

$=n.1 + cnlog_2 n = O(nlogn)$

Assume that , at step k, $T\left(\frac{n}{2^k}\right) = T(1)$

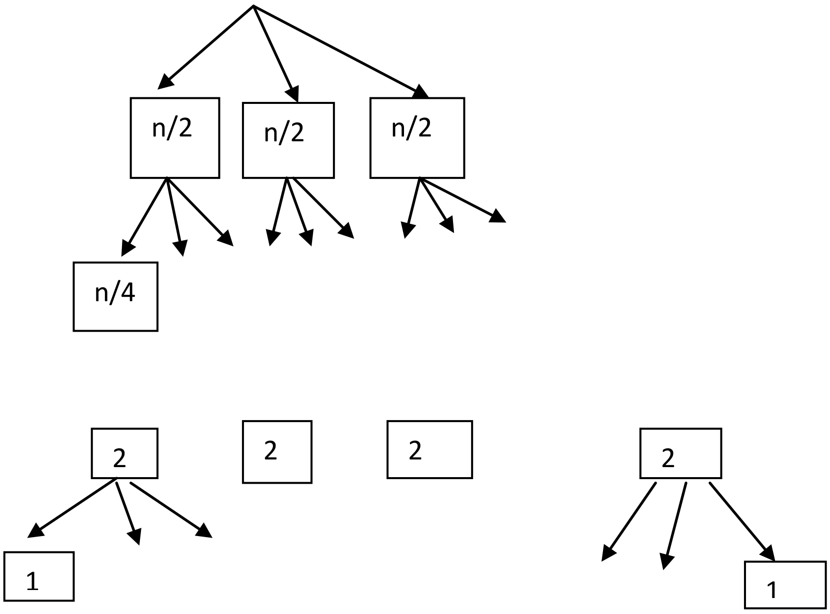
$$\left(\frac{n}{2^k}\right) = 1$$

$$2^k = n$$

k=log_2n

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

Size n



At depth k, there are 3^k sub prpblems

When prob size reduces to 1 , there is no further sub-division of the sub-problems. So, no recursive call. It may return some value which takes $O\left(\frac{n}{2^k}\right)$ if it happens at depth k. So, the sub-problem of size $\left(\frac{n}{2^k}\right)$ has only combination step. Hence no recursive calls. Only the combination step takes $O\left(\frac{n}{2^k}\right)$. The combination step is nothing but returning results of some trivial computation which takes O(1)

Running time of a sub problem at dept k is:

$T(\frac{n}{2^k})= O\left(\frac{n}{2^k}\right)$ = O(1) $\quad [\frac{n}{2^{dept\ h}} = 1 => depth = log_2$n]

Total running time at level k= $3^k \times O\left(\frac{n}{2^k}\right) = \left(\frac{3}{2}\right)^k \times O(n)$

Total running time=$\sum_{k=0}^{depth} \left(\frac{3}{2}\right)^k \times O(n)$

=O($\left(\frac{3}{2}\right)^{log_2 \mathrm{n}} \times O(n)$)=$3^{log_2 \mathrm{n}} * \frac{1}{2^{log_2 \mathrm{n}}} * O(n)$

=$O(3^{log_2 \mathrm{n}}) = O(n^{log_2 3}) = O(n^{1.59})$

Master Theorem:

T(n)=$O\left(n^d logn\right) \quad if\ d = log_b a$

$= O\left(n^d\right)\ if\ d >\ log_b a$

= $O\left(n^{log_b a}\right)$ if d<$log_b a$

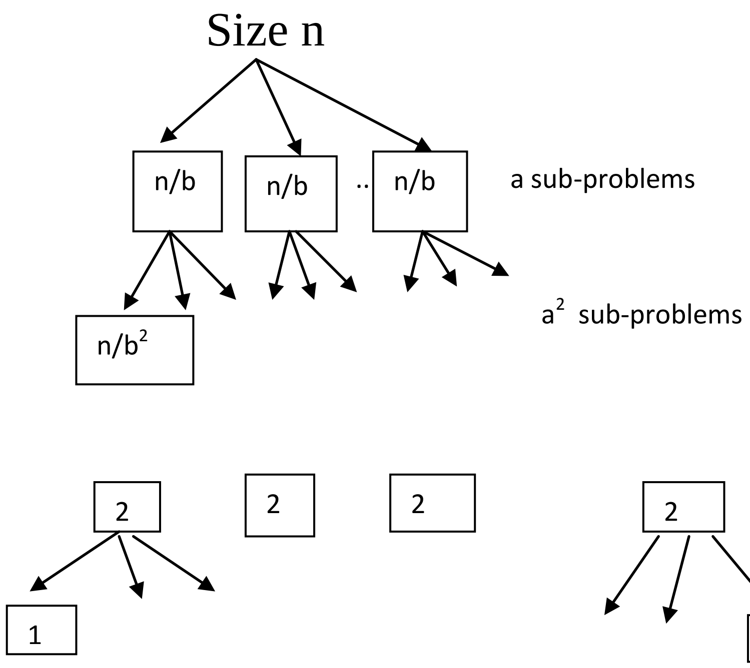
T(n)=3T(n/2)+ O(n)

a=3, b=2, d=1

T(n)= $O\left(n^{log_2 3}\right) = O(n^{1.59})$

Proof of master theorem:

T(n)=aT(n/b)+ O(n^d)



At depth k, there are a^k sub problems

Problem size at level k is $\frac{n}{b^k}$ [$\frac{n}{b^k}$ =1=> k= $log_b n$]

Running time of the prob of size $\frac{n}{b^k}$ $=O\left(\frac{n}{b^k}\right)^d$

Running time at level k=$a^k \times O\left(\frac{n}{b^k}\right)^d = \left(\frac{a}{b^d}\right)^k \times O(n^d)$

Total running time T(n) =$\sum_{k=0}^{log_b n} \left(\frac{a}{b^d}\right)^k \times O(n^d)$

case 1: $\frac{a}{b^d} < 1$=> d>$log_b a$

The series is decreasing, the first term determines the running time in big-oh

T(n)= $O(n^d)$

case 2: $\frac{a}{b^d} > 1$ ==> d<$log_b a$

The series will be increasing and the last term is considered

$$\left(\frac{a}{b^d}\right)^{log_b n} \times O(n^d)$$

$=\left(\frac{a^{log_b n}}{(b^d)^{log_b n}}\right) \times O(n^d) = \left(\frac{a^{log_b n}}{(b^{log_b n})^d}\right) \times O(n^d) = \left(\frac{a^{log_b n}}{O(n^d)}\right) \times O(n^d) = a^{log_b n} = n^{log_b a}$

T(n)=O($n^{log_b a}$)

case 3: $\frac{a}{b^d} = 1$ ===> d=$log_b a$

T(n) =$\sum_{k=0}^{log_b n} (1)^k \times O(n^d)$=O($n^d log_b n$)= O($n^d log \ \ n$)

This Proof is from the book by Sanjay Dasgupta

T(n)=2T(n/2)+ O(2^n) X

Mergesort Algorithm

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

T(n)=aT(n/b)+ O(n^d)

a=2, b=2, d=1

case1 : d>$log_b a$ 1>log 2 X

Case 2: d<$log_b a$, 1<log 2 x

case 3: d =$log_b a$, 1=log 2 , yes

T(n) =O($n^d log \ \ n$)= O($n^1 log_2 n$)= O($n \ \ log \ \ n$)

Multiplication of two n-digit numbers

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

a=3, b=2 and d=1

d<$log_b a$, 1<$log_2 3$

T(n)= $O\left(n^{log_b a}\right)$ = $O\left(n^{log_2 3}\right) = O(n^{1.59})$

Polynomial time solvable.

For any problem whose running time is O(n^k) where k is a constant and n is the input size.

bubble sort-O(n^2), Mergesort Algorithm- O(nlogn)=O(n^2)

Optimization problems--reduced to decision making problem