Algorithm

It is a finite set of instructions that if followed accomplishes a particular task. In addition all algorithms must satisfy the following criteria.

1 Input: Zero/more quantities are externally supplied.

2 Output: At least one quantity is produced

3 Definiteness: Each instruction is clear and unambiguous

4 Finiteness: If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite no. of steps.

5 Effectiveness: Every instruction must be very basic so that it can be carried out, in principle by a person using pencil and paper

The Study of algorithms includes many important and active areas of research

There are four distinct areas of study one can identify.

1 How to device algorithms

Creating an algorithm is an art which may never be fully automated. In this subject we study various design techniques that have proven to be useful in that they have often yielded good algorithms.

2 How to validate algorithm

Once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible legal inputs we refer to this process as algorithm validation.

3 How to analyse algorithms

As an algorithm is executed, it uses the computer's cpu to perform operation and its memory (both immediate and auxiliary) to hold the program and data.

Analysis of algorithm or performance analysis refers to the task of determining how much computing time and storage

an algorithm requires.

4 How to test a programs

Testing a program consist of 2 phases:

1. Debugging: Debugging is the process of executing program on sample data sets to determine whether faulty results occure and if so, to correct them

2 profiling (or performance measurment): profiling is the process of executing a correct program on data sets and measuring the time and space it takes to compute the result.

Recursive algorithm

A recursive function is a function that is defined in terms of itself.

Similarly an algorithm is said to be recursive, if the same algorithm invoked in the body.

An algorithm that call itself is direct recursive. Algorithm A is said to be indirect recursive if it calls another algorithm which in turn calls A

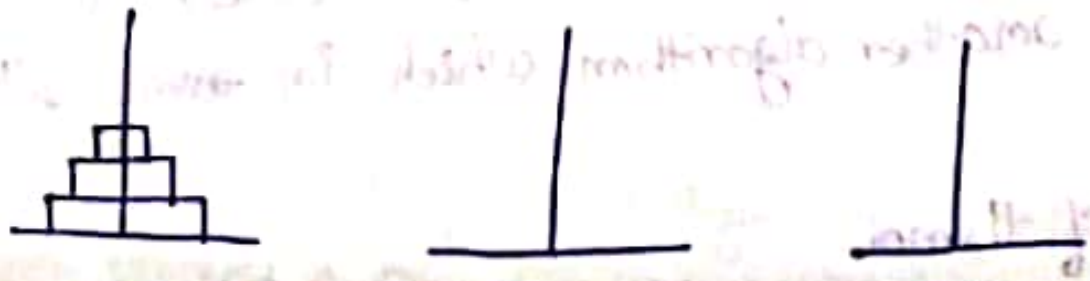
Tower of Hanoi

In the problem of TOH, there is tower called A the disks where of decreasing size and were stacked on the tower in decreasing order of size bottom to top.

Besides this tower, there were 2 other towe labelled B and C. we want to move the disk from A to B using C for intermediate storage.

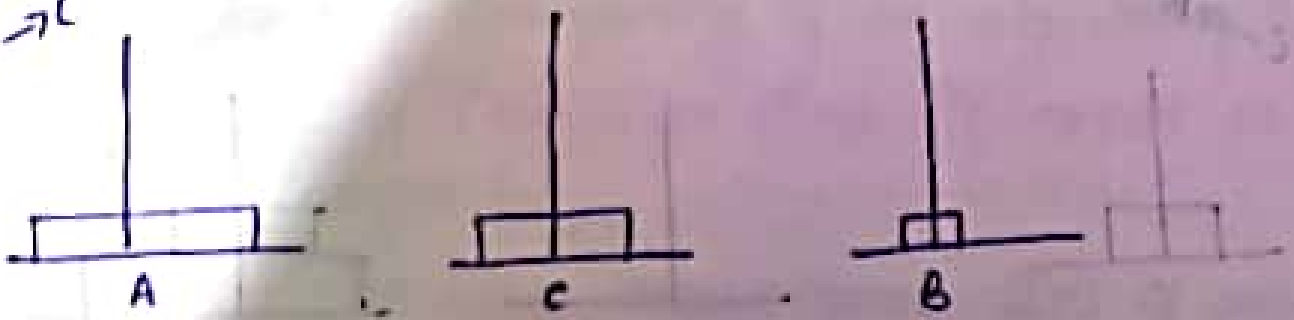
An elegent solution results from the use of recursion, assume that, the no. of disk is N. To get the largest disk to bottom of tower-B we move the remaining N-1 disk to tower-C, and then move the largest to the tower B.

Now, we left with the task of moving the disk from towerc to towerB. To do this, we have tower A and towerB available.

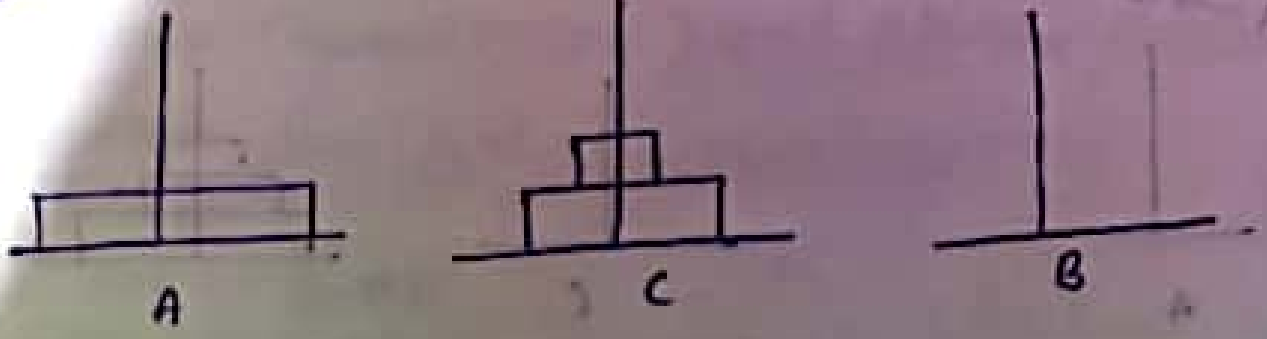


A → B

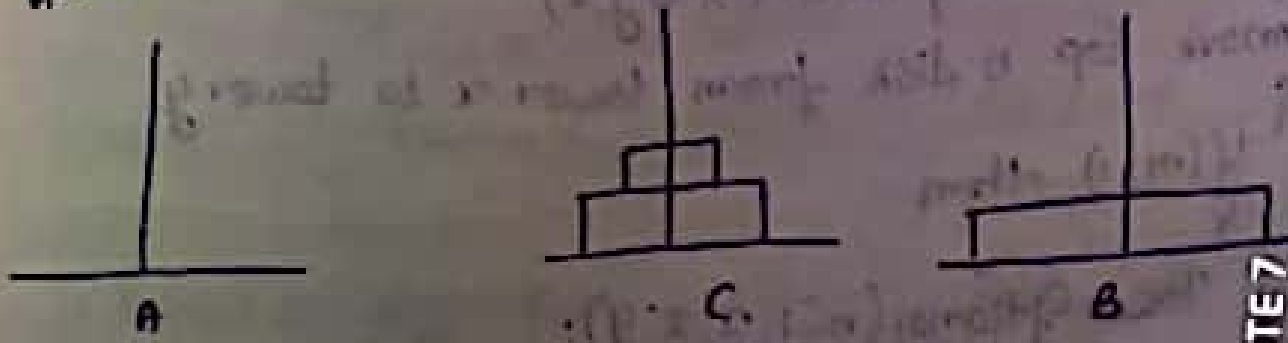
A → C



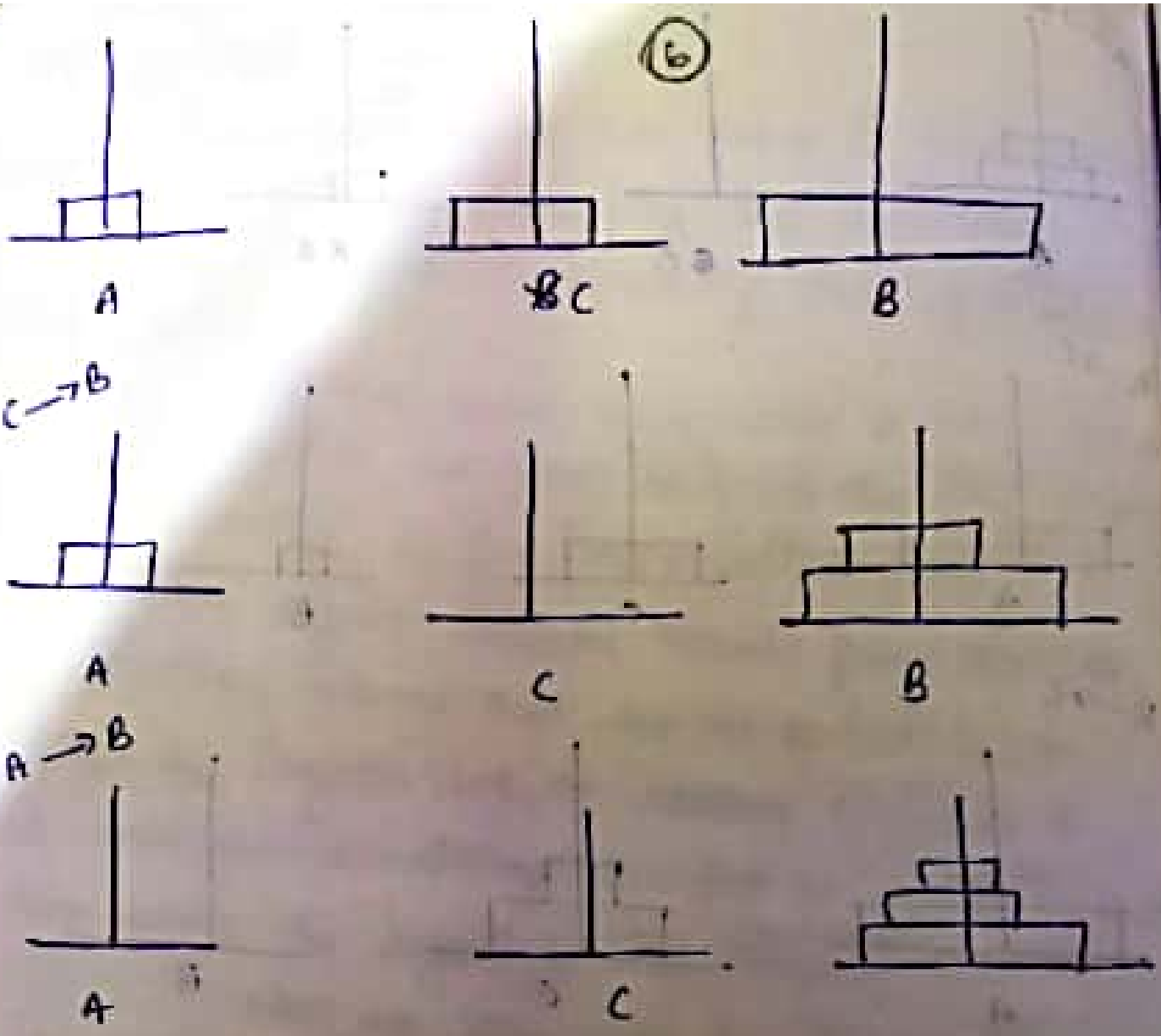
B → C



A → B



C → A



Algorithm: TowerOfHanoi(n, x, y, z)
// move top n disk from tower x to tower y
{
  if(n≥1) then
  {
    TowerOfHanoi(n-1, x, z, y);
    write("Move top disk from tower 'x' to top of tower 'y');
  } TowerOfHanoi(n-1, z, y, x);

Time and Space Complexity

The space Complexity of an algorithm is the amount of memory it needs to run to Completion.

The time Complexity of an algorithm is the amount of Cmptr time it needs to

Run to the Completion:

The Space requirement S(p) of any algorithm p may be written as S(p) = C + Sp (Instance characteristic) and C is the Constant

Algorithm Sum(a,n)

```
{
    S = 0.0;
    for i = 1 to n do
    S = S + a[i];
    return S;
}
```

S → 1
i → 1
n → 1
a[] → n
total → n+3

Ssum(n) ≥ n+6

// Iterative algorithm for Sum:

Algorithm : Rsum(a,n)

```
{
    if(n<=0)
    then return 0.0
    else
        return Rsum(a, n-1) + a[n]
}
```

// Recursive algorithm for sum

StackSpace ≥ 3(n+1)

Time Complexity

The time t(p) taken by a program p is the sum of Compile time and run time we can find the time complexity by using Counter variable 2 step Count Counter variable

Algorithm: Sum(a,n)

```
{
  Count = Count + 1 // Count is global
  S = 0.0;          It is initially zero.

  for i=1 to n do
  {
    Count = Count + 1 // for for loop
    S = S + a[i];
    Count = Count + 1 // for assignment
  }
  Count = Count + 1 // for last time of for
  Count = Count + 1 // for return
  return S;
}
```

This Simplified version of above algorithm

```
Algorithm: Sum(a,n)
{
    for i=1 to n   Count = Count+2
    Count = Count+3
}
Algorithm : Rsum(a,n)
{
    Count = Count+1;
    If n≤0 then
    {
     Count = Count+1
    return 0.0;
    }
    else
    {
    Count = Count+1;
    return Rsum(a,a-1)+a[n];
    }
}
```

When analysing a recursive program for its Step Count. we often obtain a recursive formula for Step Count

For eg:

$$t_{Rsum} = \begin{cases} 2 & ; \text{if } n = 0 \\ 2 + t_{Rsum}(n-1) & ; \text{if } n > 0 \end{cases}$$

$t_{Rsum}(n) = 2 + t_{Rsum}(n-1)$

$= 2 + 2 + t_{Rsum}(n-2)$

$= 2 + 2 + 2 + t_{Rsum}(n-3)$

$\vdots$

$= 2n + t_{Rsum}(n-n)$ $\quad t_{Rsum}(0) = 2$

$t_{Rsum} = 2n + 2$

```
Algorithm: ADD(a, b, c, m, n)
{
for i=1 to m do
{
count = count+1;
for j=1 to n do
{
count = count+1;
c[i,j]: a[i,j]+ b[i,j]
count = count+1;
}
count = count+1;
}
count = count+1;
}
```

Step count method.

A second method to determine the step count of an algorithm is to build a table in which we list the no.of steps contribute by each statement.

Stepper execution: (Se)

The Se of a statement is the amount

of which the count changes as a result of execution of last statement.

(12)

Frequency

The no of time each statement is executed

Statements

	S/e	frequency	total
Algorithm: Sum(a,n)	0	0	0
{	0	0	0
S=0.0	1	1	1
for i=1 to n do	1	n+1	n+1
S=S+a[i]	1	n	n
return S	1	1	1
}	0	0	0
			2n+3

total			
Algorithm: ADD(m,n)	0	0	0
{	0	0	0
for i=1 to m do	1	m+1	m+1
for j=1 to n do	1	m(n+1)	mn+m
c[i,j] = a[i,j] + b[i,j]	1	mn	mn
}	0	0	0
			2mn+2m+

Divide and Conquer Algorithm

Given a function to compute on 'n' inputs, the divide and conquer strategy suggest splitting the input into k distinct subsets $k \leq n$ yielding k subproblems.

Subproblems must be solved and then a method must be found to combine such solution into a solution of the whole. If the subproblems are still relatively large, then the divide and conquer strategy can possibly be re applied

Algorithm: Divide and Conquer (p)

```
{
  If Small(p) then
    return S(p);
  else
  {
    Divide p into smaller instances
    p1, p2 .... pk ; k ≥ 1
    Apply Divide and Conquer (p1), Divide and
    Conquer (p2) .... Divide and Conquer (pk);
    return (Combine (Divide and Conquer(p1), Divide and Conquer(p2) .... Divide and Conquer(pk)));
  }
}
```

Finding minimum and maximum

The devide and Conquer algorithm for finding the max and minimum

Let p=n, a[1] ... a[j] and we are trying to find the max and min of this list

Let Small(p) will be true, if n≤2

If the list has more than 2 elements we right devide p into 2 instances

$p_1 = ([n/2], a[1] \cdots a[n/2])$

$p_2 = [n - n/2], a[n/2+1], a[n/2+2] \cdots a[n]$

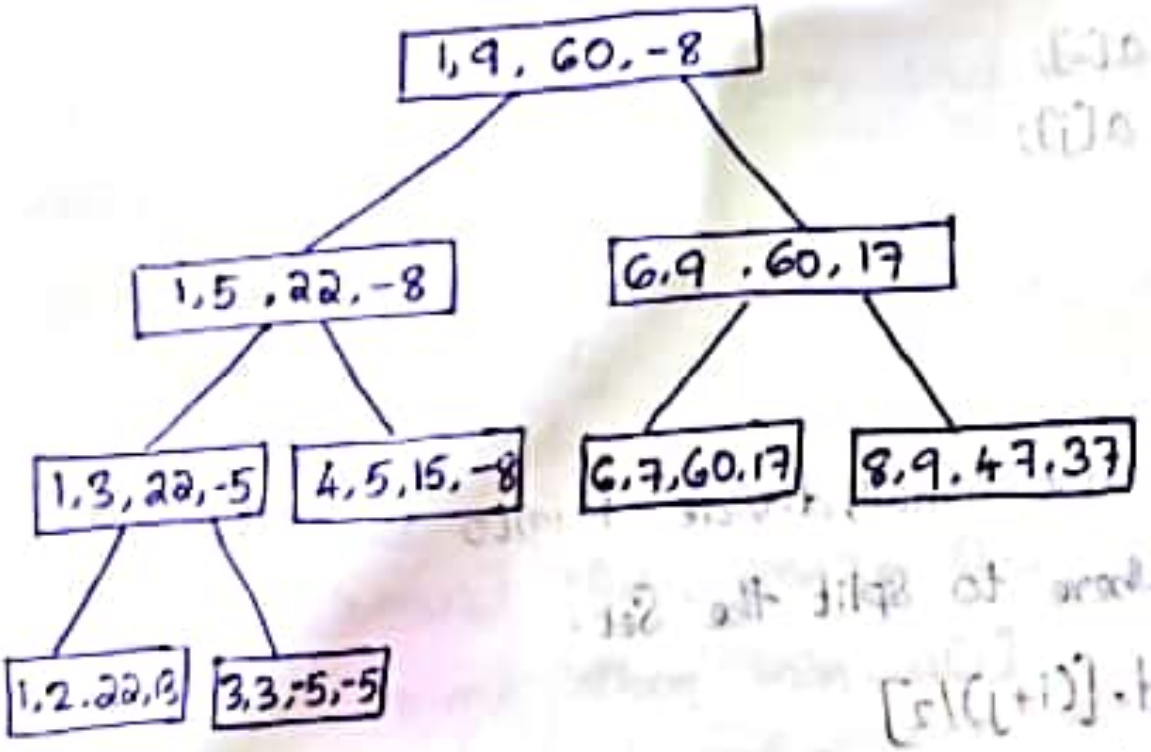
After dividing p into Small Sub problems, we can Solve them bop. Recursively invoking the Same devide and Conques algorithm.

If max(p) and min(p) are the max and min of the elements in p. Then max[p] is the larger of max(p1) and max(p2) also min(p) is the Smaller of min(p1) and min(p2)

a:

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
22	13	-5	-8	15	60	17	37	47

15



Algorithm: maxmin(i,j,max,min)
// a[1:n] is a global array. Parameters are i and j
// are integers, 1≤i≤j≤n. The effect is two sets
// max and min to the largest and smallest values
// in a[i:j] respectively.

```
{
  if (i=j) then max=min=a[i] // Small(P)
  else if (i=j-1) then // Another Case of Small(P)
  {
    if (a[i]<a[j]) then
    {
      max=a[j];
```

```
    min = a[i];
  }
  else
  {
    max = a[i];
    min = a[j];
  }
}
else
{
// if P is not Small, divide P into Subproblems.
// find where to Split the Set.
    mid = [(i+j)/2]
 // Solve the Subproblems...
  MaxMin(i, mid, max, min);
  MaxMin(mid+1, j, max1, min1);
  // Combine the Solutions.
  if (max < max1) then max = max1;
  if (min > min1) then min = min1;
 }
}
```

Straight forward maximum and minimum

17

Algorithm: StraightMaxMin(a,n, max, min)
// Set max to the maximum and min to the
// minimum of a[1:n].
{
max := min := a[1];
for i=2 to n do
{
if(a[i]>max) then max := a[i].
if(a[i]< min) then min = a[i]
}
}

worst(avg) is O(n log n)

MergeSort

Given a Sequence of n elements (also called keys). a[1], a[2] ...., a[n], the general idea is to imagine them split into 2 sets, a[1] ... a[n/2] and a[n/2+1], a[n/2+2] .... a[n].

Each Set is individually Sorted, and resulting Sorted Sequenceds are merged to produce a Single Sorted Sequence of n elements.

MergeSort describes this process using recursion and a function, merge. which merges two Sorted Sets.

Consider the array of 10 elements a[1] - a[10]
a[1:10] (310, 285, 179, 652, 351, 423, 861, 254, 450, 520)

Algorithm mergesort begins by Splitting a[] into 2 Subarrays. Each of Size 5, a[1:5] and a[6:10]. The elements in a[1:5] are Split into 2 Sub arrays a[1:3] and a[4:5] and So on. Finally we get one element in Subarrays now the merging begins.

(310 | 285 | 179 | 652, 351 | 423, 861 | 254, 450, 520)

1st Sublist 285, 310 | 179 (652 22)

310, 285, 179, 652, 351

2nd Sublist

423, 861, 254, 450, 520

19

310, 285, 179 | 652, 351 | 423, 861, 254 | 450, 520

310, 285 | 179 | 652 | 351 | 423, 861 | 254 | 450 | 520

310 | 285 | 179 | 652 | 351 | 423 | 861 | 254 | 450 | 520

285 | 310 | 179 | 652 | 351 | 423 | 861 | 254 | 450 | 520

179 | 285 | 310 | 652 | 351 | 423 | 861 | 254 | 450 | 520

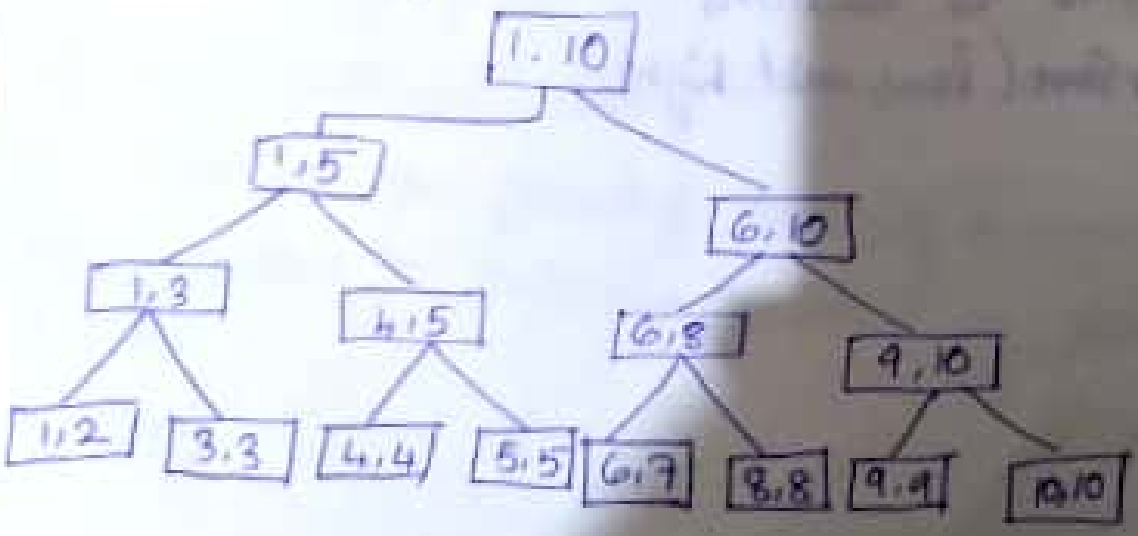
179 | 285 | 310 | 351 | 652 | 423 | 861 | 254 | 450 | 520

179 | 285 | 310 | 351 | 652 | 254 | 423 | 861 | 450 | 520

179 | 285 | 310 | 351 | 652 | 254 | 423 | 450 | 861 | 520

179 | 285 | 310 | 351 | 652 | 254 | 423 | 450 | 520 | 861.

Tree of calls of sort of 1:10



```
Algorithm MergeSort(low, high)
// a[low:high] is a global array to be Sorted. Small(p)
// is true if there is only one element to sort. In
// this case, the list is already sorted.
{
  if (low < high) then // if there are more than one
                       element
  {
    // divide P into Subproblems
    // find where to split the set.
    mid := [(low + high)/2];
    // Solve the Subproblems
    MergeSort(low, mid);
    MergeSort(mid+1, high);
    //Combine the Solutions.
    MergeSort(low, mid, high);
  }
}
```

QuickSort

QuickSort is accomplished by rearranging the elements in $a[1:n]$ such that $a[i] \leq a[j]$ $\forall i$ between 1 and m and $\forall j$ between m+1 and n for some m, $1 \leq m \leq n$. Thus the elements in $a[1:m]$ and $a[m+1:n]$ can be independently sorted. No merge is needed

The rearrangment of elements is accomplished by picking some element of $a[]$, say $t = a[s]$ and then reordering the other elements so that all elements appearing before t in $a[1:n]$ are less than or equal to t and all elements appearing after t are greater than or equal to t. Thus rearranging is referred to as partitioning.

eg: How partition works

Consider the following array of n elements. The function is initially invoked as partition(a, 1, n)

The element a[1] = 65 is the partitioning element and it is eventually (on the 6th row) to be the 5th smallest element of the set.

Note that, the remaining elements are unsorted but partitioned about a[5]=55

1	2	3	4	5	6	7	8	9	10	i	j
65	70	75	30	85	60	55	50	45		2	9
65	45	75	30	85	60	55	50	70		3	8
65	45	50	30	85	60	55	75	70		4	7
65	45	50	55	85	60	30	75	70		5	6
65	45	50	55	60	85	30	75	70		6	5
60	45	50	55	65	85	30	75	70			

Algorithm : partition (a, m, p)
// within a[m] [m+1] .... a[p-1] the elements are rearranged in such a manner that if initially t = a[m] then after completion a[q] = t for some q btw m and p-1, a[k] ≤ t for m ≤ k ≤ q and a[k] ≥ t for q < k < p. q is returned // set a[p] = ∞
{
V = a[m]
i = m, j = p
Repeat
{
repeat
i = i + 1
until (a[i] ≥ v);
Repeat
j = j + 1
until (a[j] ≤ v);
If (i < j) then Interchange (a, i, j)
}
until (i ≥ j)
a[m] = a[j];

Return j;

}

Algorithm : Interchange(a, i, j)

// Exchange a[i] with a[j]

{

p = a[i];

a[i] = a[j];

a[j] = p;

}

Algorithm : QuickSort(p, q)

// Sorts the elements a[p] . . . . . a[q] with respect to the global array a[1:n] into ascending order. a[n+1] is considered to be defined and must be greater than or equal to all the elements in a[1:n]

{

// divide P into SubProblems

j = Partition(a, p, q+1)

// j is the position of the partitioning elements

// Solve the SubProblems.

QuickSort(p, j-1);
QuickSort(j+1, q);
// There is no need for combining solutions
}
}

Selection

The ~~partior~~ partitioning algorithm can also be used to obtain an efficient solution for the selection problems. In this problem, we are given n elements A[1:n] and are required to determine the kth smallest element.

If the partitioning element, v is positioned at A[j], then, j-1 elements are less than or equal to A[j] and n-j elements are greater than or equal to A[j]. Hence

If k<j, the kth smallest element is in A[1:j-1]

If k==j, then A[j] is the kth smallest element.

If $k > j$, then k^{th} smallest element $[k-j]^{th}$ is en $a[j+1:n]$

Algorithm: Select(a, n, k)

// Selects the k^{th} smallest element in $a[1:n]$ and

// places it en the k^{th} position of $a[]$. Then remaining elements are rearranged.

// $a[m] \leq a[k]$; for $1 \leq m \leq k$

// $a[m] \geq a[k]$; for $k < m \leq n$

{

low = 1; up = n+1;

a[n+1] = ∞;

repeat

{

// Each time the loop is entered.

// $1 \leq low \leq k \leq up \leq n+1$

j = Partition(a, low, up);

// j is a[j] is the j^{th} smallest value en a[]

if(k==j) then return;

else

if(k<j) then up=j; //j is the new upper limit.

else

low=j+1; //j+1 is the new lower limit

until(false);

}

Consider the line 'e' command with A[10]=+∞

65 70 75 30 35 60 55 50 45 +∞

If k=5, then the 1st call of partition will be Self recurrent.

Since 65 is replaced into A[5]

Let k=7, the next invocation of partitioning is partition(6,10)

65 35 30 75 70 +∞

65 70 30 75 35 +∞

The next invocation is partition(6,9)

65 35 30 75 70 +∞

65 70 30 75 35 +∞

This time the 6th element has been found partition (7 19)

65 70 80 75 85 to

65 70 75 80 85 to

It needs one more call partition (7 18) this program only an Interchange between A[7] and A[8]. In analysing Select, we make the same assumption that when we made for quick sort

1 The n element are distinct

2 The i/p distribution is such that the partition element can be the ith smallest element of A[p:q] with an equal probability for each i, $1 \le i \le p-m$

Strassen's Matrix Multiplication

Let A and B be 2 nxn matrices. The product matrix C=AB is also an nxn matrix whose (i,j)th element is formed by taking the element in ith row of A and jth column of B and multiplying them to get,

$$i,j = \sum_{1 \le k \le n} A(i,k)\ B(k,j) \quad \forall\ i \& j \text{ between } 1 \& n.$$

The divide and Conquer strategy suggest another way of computing product of 2 nxn matrices. For simplicity we assume that n is power of 2. i.e, there must a non-negative Integer k. Such that $n = 2^k$.

In the case n is not power of 2, then enough rows and columns of 0's can be added to both A & B so that the resulting dimensions are power of 2.

Let A and B are 2 matrices and AB can be calculated as

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

$C_{11} = A_{11}B_{11} + A_{12}B_{21}$

$C_{12} = A_{11}B_{12} + A_{12}B_{22}$ ——— (1)

$C_{21} = A_{21}B_{11} + A_{22}B_{21}$

$C_{22} = A_{21}B_{12} + A_{22}B_{22}$

Imagine that the above matrices A and B are each partitioned into 4 square submatrices, each submatrix having dimension $n/2 \times n/2$.

If $n=2$, then the above formulas are computed using multiplication operation for the elements of A_{kk}.

For $n>2$, the elements of C can be computed using matrix multiplication and addition operations applied to matrices of size $n/2 \times n/2$ matrices and 4 addition

Volker Strassen has discovered a way to compute the C_{ij}'s of (1) using only 7 multiplication and 18 additions or subtractions. It is method involves first computing the 7 $n/2 \times n/2$ matrices P, Q, R, S, T, U and V and then

$P = (A_{11} + A_{22})(B_{11} + B_{22})$

$Q = (A_{21} + A_{22})B_{11}$

$R = A_{11}(B_{12} - B_{22})$

$S = A_{22}(B_{21} - B_{11})$

$T = (A_{11} + A_{12})B_{22}$

$U = (A_{21} - A_{11})(B_{11} + B_{12})$

$V = (A_{12} - A_{22})(B_{21} + B_{22})$

Then C_{ij}'s computed as

$C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U$

Consider the eg.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 5+14 & 6+16 \\ 15+28 & 18+32 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

$P = (A_{11} + A_{22})(B_{11} + B_{22}) = 5 \times 13 = 65$

$Q = (A_{21} + A_{22})B_{11} = 7 \times 5 = 35$

$R = A_{11}(B_{12} - B_{22}) = 1 \times -2 = -2$

$S = A_{22}(B_{21} - B_{11}) = 4 \times 2 = 8$

$T = (A_{11} + A_{12})B_{22} = 3 \times 2 = 6$

$U = (A_{21} - A_{11})(B_{11} + B_{12}) = 2 \times 11 = 22$

$V = (A_{12} - A_{22})(B_{21} + B_{22}) = -2 \times 15 = -30$

$C_{11} = 9$

$C_{12} = 22$

$C_{21} = 43$

$C_{22} = 50$

$$\begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

//