Chapter 3

Divide and Conquer

General Method

Divide and conquer is a design strategy which is well known to breaking down efficiency barriers. When the method applies, it often leads to a large improvement in time complexity. For example, from O (n^2) to O (n log n) to sort the elements.

Divide and conquer strategy is as follows: divide the problem instance into two or more smaller instances of the same problem, solve the smaller instances recursively, and assemble the solutions to form a solution of the original instance. The recursion stops when an instance is reached which is too small to divide. When dividing the instance, one can either use whatever division comes most easily to hand or invest time in making the division carefully so that the assembly is simplified.

Divide and conquer algorithm consists of two parts:

Divide :	Divide the problem into a number of sub problems. The sub problems are solved recursively.
Conquer :	The solution to the original problem is then formed from the solutions to the sub problems (patching together the answers).

Traditionally, routines in which the text contains at least two recursive calls are called divide and conquer algorithms, while routines whose text contains only one recursive call are not. Divide–and–conquer is a very powerful use of recursion.

Control Abstraction of Divide and Conquer

A control abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined. The control abstraction for divide and conquer technique is DANDC(P), where P is the problem to be solved.

```
DANDC (P)
{
        if SMALL (P) then return S (p);
        else
        {
                divide p into smaller instances $p_1, p_2, …. P_k, k \geq 1$;
                apply DANDC to each of these sub problems;
                return (COMBINE (DANDC ($p_1$) , DANDC ($p_2$),…., DANDC ($p_k$));
        }
}
```

SMALL (P) is a Boolean valued function which determines whether the input size is small enough so that the answer can be computed without splitting. If this is so function 'S' is invoked otherwise, the problem 'p' into smaller sub problems. These sub problems $p_1, p_2, . . . , p_k$ are solved by recursive application of DANDC.

If the sizes of the two sub problems are approximately equal then the computing time of DANDC is:

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ 2T(n/2) + f(n) & \text{otherwise} \end{cases}$$

Where, T (n) is the time for DANDC on 'n' inputs
g (n) is the time to complete the answer directly for small inputs and
f (n) is the time for Divide and Combine

Binary Search

If we have 'n' records which have been ordered by keys so that $x_1 < x_2 < \dots < x_n$. When we are given a element 'x', binary search is used to find the corresponding element from the list. In case 'x' is present, we have to determine a value 'j' such that a[j] = x (successful search). If 'x' is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key a[mid], and compare 'x' with a[mid]. If x = a[mid] then the desired record has been found. If x < a[mid] then 'x' must be in that portion of the file that precedes a[mid], if there at all. Similarly, if a[mid] > x, then further search is only necessary in that past of the file which follows a[mid]. If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every un-successful comparison of 'x' with a[mid] will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved often each comparison between 'x' and a[mid], and since an array of length 'n' can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$

Algorithm Algorithm

```
BINSRCH (a, n, x)
//      array a(1 : n) of elements in increasing order, n ≥0,
//      determine whether 'x' is present, and if so, set j such that x = a(j)
//      else return j

{
        low :=1 ; high :=n ;
        while (low ≤ high) do
        {
                mid :=|(low + high)/2|
                if (x < a [mid]) then high:=mid – 1;
                else if (x > a [mid]) then low:= mid + 1
                        else return mid;
        }
        return 0;
}
```

low and *high* are integer variables such that each time through the loop either 'x' is found or *low* is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if 'x' is not present.

Example for Binary Search

Let us illustrate binary search on the following 9 elements:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Elements	-15	-6	0	7	9	23	54	82	101

The number of comparisons required for searching different elements is as follows:

1. Searching for x = 101

low	high	mid
1	9	5
6	9	7
8	9	8
9	9	9
		found

Number of comparisons = 4

2. Searching for x = 82

low	high	mid
1	9	5
6	9	7
8	9	8
		found

Number of comparisons = 3

3. Searching for x = 42

low	high	mid
1	9	5
6	9	7
6	6	6
7	6	not found

Number of comparisons = 4

4. Searching for x = -14

low	high	mid
1	9	5
1	4	2
1	1	1
2	1	not found

Number of comparisons = 3

Continuing in this manner the number of element comparisons needed to find each of nine elements is:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Elements	-15	-6	0	7	9	23	54	82	101
Comparisons	*3*	*2*	*3*	*4*	*1*	*3*	*2*	*3*	*4*

No element requires more than 4 comparisons to be found. Summing the comparisons needed to find all nine items and dividing by 9, yielding 25/9 or approximately 2.77 comparisons per successful search on the average.

There are ten possible ways that an un-successful search may terminate depending upon the value of x.

If $x < a[1]$, $a[1] < x < a[2]$, $a[2] < x < a[3]$, $a[5] < x < a[6]$, $a[6] < x < a[7]$ or $a[7] < x < a[8]$ the algorithm requires 3 element comparisons to determine that 'x' is not present. For all of the remaining possibilities BINSRCH requires 4 element comparisons. Thus the average number of element comparisons for an unsuccessful search is:

$(3 + 3 + 3 + 4 + 4 + 3 + 3 + 3 + 4 + 4) / 10 = 34/10 = 3.4$

The time complexity for a successful search is $O(\log n)$ and for an unsuccessful search is $\Theta(\log n)$.

Successful searches			**un-successful searches**
$\Theta(1)$,	$\Theta(\log n)$,	$\Theta(\log n)$	$\Theta(\log n)$
Best	average	worst	best, average and worst

Analysis for worst case

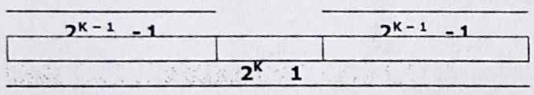
Let T (n) be the time complexity of Binary search

The algorithm sets mid to [n+1 / 2]

Therefore,

$$
\begin{aligned}
T(0) &= 0 & \\
T(n) &= 1 & \text{if } x = a\,[mid] \\
&= 1 + T([(n + 1) / 2] - 1) & \text{if } x < a\,[mid] \\
&= 1 + T(n - [(n + 1)/2]) & \text{if } x > a\,[mid]
\end{aligned}
$$

Let us restrict 'n' to values of the form $n = 2^K - 1$, where 'k' is a non-negative integer. The array always breaks symmetrically into two equal pieces plus middle element.



Algebraically this is $\left\lceil \dfrac{n+1}{2} \right\rceil = \left\lceil \dfrac{2^K - 1 + 1}{2} \right\rceil = 2^{K-1}$ for $K \geq 1$

Giving,

$$
\begin{aligned}
T(0) &= 0 & \\
T(2^k - 1) &= 1 & \text{if } x = a\,[mid] \\
&= 1 + T(2^{K-1} - 1) & \text{if } x < a\,[mid] \\
&= 1 + T(2^{k-1} - 1) & \text{if } x > a\,[mid]
\end{aligned}
$$

In the worst case the test x = a[mid] always fails, so

$w(0) = 0$

$w(2^k - 1) = 1 + w(2^{k-1} - 1)$

This is now solved by repeated substitution:

$$\begin{aligned} w(2^k - 1) &= 1 + w(2^{k-1} - 1) \\ &= 1 + [1 + w(2^{k-2} - 1)] \\ &= 1 + [1 + [1 + w(2^{k-3} - 1)]] \\ &= \dots\dots\dots \\ &= \dots\dots\dots \\ &= i + w(2^{k-i} - 1) \end{aligned}$$

For $i \leq k$, letting $i = k$ gives $w(2^k - 1) = K + w(0) = k$

But as $2^K - 1 = n$, so $K = \log_2(n + 1)$, so

$$w(n) = \log_2(n + 1) = O(\log n)$$

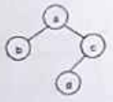
for $n = 2^K - 1$, concludes this analysis of binary search.

Although it might seem that the restriction of values of 'n' of the form $2^K - 1$ weakens the result. In practice this does not matter very much, w(n) is a monotonic increasing function of 'n', and hence the formula given is a good approximation even when 'n' is not of the form $2^K - 1$.

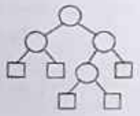
External and Internal path length:

The lines connecting nodes to their non-empty sub trees are called edges. A non-empty binary tree with n nodes has n−1 edges. The size of the tree is the number of nodes it contains.

When drawing binary trees, it is often convenient to represent the empty sub trees explicitly, so that they can be seen. For example:

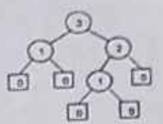


The tree given above in which the empty sub trees appear as square nodes is as follows:

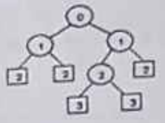


The square nodes are called as external nodes E(T). The square node version is sometimes called an extended binary tree. The round nodes are called internal nodes I(T). A binary tree with n internal nodes has n+1 external nodes.

The height h(x) of node 'x' is the number of edges on the longest path leading down from 'x' in the extended tree. For example, the following tree has heights written inside its nodes:



The depth d(x) of node 'x' is the number of edges on path from the root to 'x'. It is the number of internal nodes on this path, excluding 'x' itself. For example, the following tree has depths written inside its nodes:



The internal path length I(T) is the sum of the depths of the internal nodes of 'T':

$$I(T) = \sum_{x \in I(T)} d(x)$$

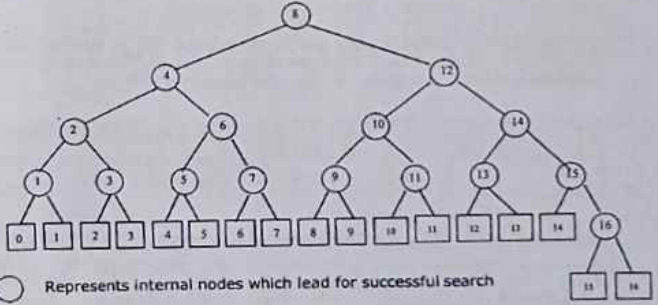
The external path length E(T) is the sum of the depths of the external nodes:

$$E(T) = \sum_{x \in E(T)} d(x)$$

For example, the tree above has I(T) = 4 and E(T) = 12.

A binary tree T with 'n' internal nodes, will have I(T) + 2n = E(T) external nodes.

A binary tree corresponding to binary search when n = 16 is



○ Represents internal nodes which lead for successful search

□ External square nodes, which lead for unsuccessful search.

Let C_N be the average number of comparisons in a successful search.

C'_N be the average number of comparison in an un successful search.

Then we have,

$$C_N = 1 + \frac{\text{internal path length of tree}}{N}$$

$$C'_N = \frac{\text{External path length of tree}}{N+1}$$

$$C_N = \left(1 + \frac{1}{N}\right) C'_N - 1$$

External path length is always 2N more than the internal path length.

Merge Sort

Merge sort algorithm is a classic example of divide and conquer. To sort an array, recursively, sort its left and right halves separately and then merge them. The time complexity of merge mort in the *best case, worst case* and *average case* is O(n log n) and the number of comparisons used is nearly optimal.

This strategy is so simple, and so efficient but the problem here is that there seems to be no easy way to merge two adjacent sorted arrays together in place (The result must be build up in a separate array).

The fundamental operation in this algorithm is merging two sorted lists. Because the lists are sorted, this can be done in one pass through the input, if the output is put in a third list.

The basic merging algorithm takes two input arrays 'a' and 'b', an output array 'c', and three counters, *a ptr, b ptr* and *c ptr,* which are initially set to the beginning of their respective arrays. The smaller of *a[a ptr]* and *b[b ptr]* is copied to the next entry in 'c', and the appropriate counters are advanced. When either input list is exhausted, the remainder of the other list is copied to 'c'.

To illustrate how merge process works. For example, let us consider the array 'a' containing 1, 13, 24, 26 and 'b' containing 2, 15, 27, 38. First a comparison is done between 1 and 2. 1 is copied to 'c'. Increment *a ptr* and *c ptr.*

1	2	3	4
1	13	24	26
h *ptr*			

5	6	7	8
2	15	27	28
j *ptr*			

1	2	3	4	5	6	7	8
1							
i *ptr*							

and then 2 and 13 are compared. 2 is added to 'c'. Increment *b ptr* and *c ptr.*

1	2	3	4
1	13	24	26
	h *ptr*		

5	6	7	8
2	15	27	28
j *ptr*			

1	2	3	4	5	6	7	8
1	2						
	i *ptr*						

then 13 and 15 are compared. 13 is added to 'c'. Increment *a ptr* and *c ptr.*

1	2	3	4
1	13	24	26
	h *ptr*		

5	6	7	8
2	15	27	28
	j *ptr*		

1	2	3	4	5	6	7	8
1	2	13					
		i *ptr*					

24 and 15 are compared. 15 is added to 'c'. Increment *b ptr* and *c ptr.*

1	2	3	4
1	13	24	26
		h *ptr*	

5	6	7	8
2	15	27	28
	j *ptr*		

1	2	3	4	5	6	7	8
1	2	13	15				
			i *ptr*				

24 and 27 are compared. 24 is added to 'c'. Increment *a ptr* and *c ptr.*

1	2	3	4
1	13	24	26
		h *ptr*	

5	6	7	8
2	15	27	28
		j *ptr*	

1	2	3	4	5	6	7	8
1	2	13	15	24			
				i *ptr*			

26 and 27 are compared. 26 is added to 'c'. Increment *a ptr* and *c ptr.*

1	2	3	4
1	13	24	26
			h *ptr*

5	6	7	8
2	15	27	28
		j *ptr*	

1	2	3	4	5	6	7	8
1	2	13	15	24	26		
					i *ptr*		

As one of the lists is exhausted. The remainder of the b array is then copied to 'c'.

1	2	3	4
1	13	24	26

h *ptr*

5	6	7	8
2	15	27	28
		j *ptr*	

1	2	3	4	5	6	7	8
1	2	13	15	24	26	27	28

i *ptr*

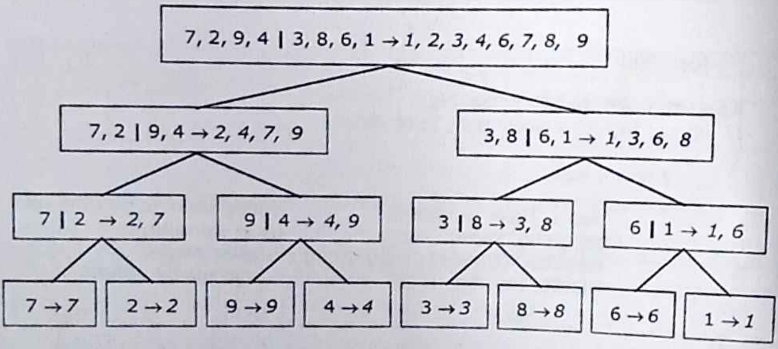
Algorithm

```
Algorithm MERGESORT (low, high)
// a (low : high) is a global array to be sorted.
{
        if (low < high)
        {
                mid := ⌊(low  + high)/2⌋          //finds where to split the set
                MERGESORT(low,  mid)              //sort one subset
                MERGESORT(mid+1, high)  //sort the other subset
                MERGE(low,  mid,  high)           // combine the results
        }
}
```

```
Algorithm MERGE (low, mid, high)
// a (low : high) is a global array containing two sorted subsets
// in a (low : mid) and in a (mid + 1 : high).
// The objective is to merge these sorted sets into single sorted
// set residing in a (low : high). An auxiliary array B is used.
{
        h :=low; i := low; j:= mid + 1;
        while ((h ≤ mid) and (J ≤ high)) do
        {
                if (a[h] ≤ a[j]) then
                {
                        b[i] := a[h]; h := h + 1;
                }
                else
                {
                        b[i] :=a[j]; j := j + 1;
                }
                i := i + 1;
        }
        if (h > mid) then
                for k := j to high do
                {
                        b[i] := a[k]; i := i + 1;
                }
        else
                for k := h to mid do
                {
                        b[i]  := a[K];  i := i + l;
                }
        for k := low to high do
                a[k] := b[k];
}
```

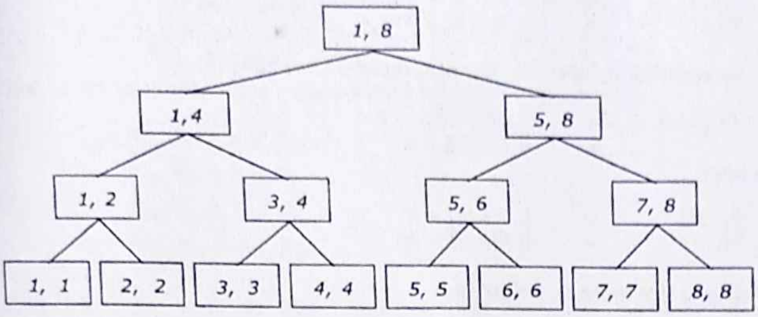
Example

For example let us select the following 8 entries 7, 2, 9, 4, 3, 8, 6, 1 to illustrate merge sort algorithm:



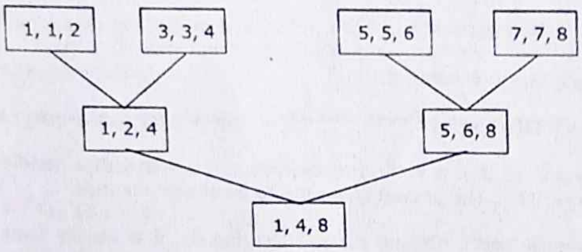
Tree Calls of MERGESORT(1, 8)

The following figure represents the sequence of recursive calls that are produced by MERGESORT when it is applied to 8 elements. The values in each node are the values of the parameters low and high.



Tree Calls of MERGE()

The tree representation of the calls to procedure MERGE by MERGESORT is as follows:



Analysis of Merge Sort

We will assume that 'n' is a power of 2, so that we always split into even halves, so we solve for the case $n = 2^k$.

For n = 1, the time to merge sort is constant, which we will be denote by 1. Otherwise, the time to merge sort 'n' numbers is equal to the time to do two recursive merge sorts of size n/2, plus the time to merge, which is linear. The equation says this exactly:

$$T(1) = 1$$
$$T(n) = 2\,T(n/2) + n$$

This is a standard recurrence relation, which can be solved several ways. We will solve by substituting recurrence relation continually on the right-hand side.

We have, $T(n) = 2T(n/2) + n$

Since we can substitute n/2 into this main equation

$$\begin{aligned} 2\,T(n/2) &= 2\,(2\,(T(n/4)) + n/2) \\ &= 4\,T(n/4) + n \end{aligned}$$

We have,

$$\begin{aligned} T(n/2) &= 2\,T(n/4) + n \\ T(n) &= 4\,T(n/4) + 2n \end{aligned}$$

Again, by substituting n/4 into the main equation, we see that

$$\begin{aligned} 4T\,(n/4) &= 4\,(2T(n/8)) + n/4 \\ &= 8\,T(n/8) + n \end{aligned}$$

So we have,

$$\begin{aligned} T(n/4) &= 2\,T(n/8) + n \\ T(n) &= 8\,T(n/8) + 3n \end{aligned}$$

Continuing in this manner, we obtain:

$$T(n) = 2^k\,T(n/2^k) + K.\,n$$

As $n = 2^k$, $K = \log_2 n$, substituting this in the above equation

$$\begin{aligned} T(n) &= 2^{\log_2 n}\,T\left(\frac{2^k}{2^k}\right) + \log_2 n\,.\,n \\ &= n\,T(1) + n \log n \\ &= n \log n + n \end{aligned}$$

Representing this in O notation:

$$T(n) = \mathbf{O(n \log n)}$$

We have assumed that $n = 2^k$. The analysis can be refined to handle cases when 'n is not a power of 2. The answer turns out to be almost identical.

Although merge sort's running time is O(n log n), it is hardly ever used for main memory sorts. The main problem is that merging two sorted lists requires linear extra memory and the additional work spent copying to the temporary array and back, throughout the algorithm, has the effect of slowing down the sort considerably. *The Best and worst case time complexity of Merge sort is O(n log n).*

Strassen's Matrix Multiplication:

The matrix multiplication of algorithm due to Strassens is the most dramatic example of divide and conquer technique (1969).

The usual way to multiply two n x n matrices A and B, yielding result matrix 'C' as follows :

```
for i := 1 to n do
        for j :=1 to n do
                c[i, j] := 0;
                for K: = 1 to n do
                        c[i, j] := c[i, j] + a[i, k] * b[k, j];
```

This algorithm requires n^3 scalar multiplication's (i.e. multiplication of single numbers) and n^3 scalar additions. So we naturally cannot improve upon.

We apply divide and conquer to this problem. For example let us considers three multiplication like this:

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

Then c_{ij} can be found by the usual matrix multiplication algorithm,

$$C_{11} = A_{11} . B_{11} + A_{12} . B_{21}$$
$$C_{12} = A_{11} . B_{12} + A_{12} . B_{22}$$
$$C_{21} = A_{21} . B_{11} + A_{22} . B_{21}$$
$$C_{22} = A_{21} . B_{12} + A_{22} . B_{22}$$

This leads to a divide–and–conquer algorithm, which performs nxn matrix multiplication by partitioning the matrices into quarters and performing eight (n/2)x(n/2) matrix multiplications and four (n/2)x(n/2) matrix additions.

$$T(1) = 1$$
$$T(n) = 8\,T(n/2)$$

Which leads to $T(n) = O(n^3)$, where n is the power of 2.

Strassens insight was to find an alternative method for calculating the C_{ij}, requiring seven (n/2) x (n/2) matrix multiplications and eighteen (n/2) x (n/2) matrix additions and subtractions:

$$P = (A_{11} + A_{22})\ (B_{11} + B_{22})$$
$$Q = (A_{21} + A_{22})\ B_{11}$$
$$R = A_{11}\ (B_{12} - B_{22})$$
$$S = A_{22}\ (B_{21} - B_{11})$$
$$T = (A_{11} + A_{12})\ B_{22}$$
$$U = (A_{21} - A_{11})\ (B_{11} + B_{12})$$
$$V = (A_{12} - A_{22})\ (B_{21} + B_{22})$$
$$C_{11} = P + S - T + V$$
$$C_{12} = R + T$$
$$C_{21} = Q + S$$
$$C_{22} = P + R - Q + U.$$

This method is used recursively to perform the seven (n/2) x (n/2) matrix multiplications, then the recurrence equation for the number of scalar multiplications performed is:

$$
\begin{aligned}
T(1) &= 1 \\
T(n) &= 7\,T(n/2)
\end{aligned}
$$

Solving this for the case of $n = 2^k$ is easy:

$$
\begin{aligned}
T(2^k) &= 7\,T(2^{k-1}) \\
&= 7^2\,T(2^{k-2}) \\
&= \text{------} \\
&= \text{------} \\
&= 7^i\,T(2^{k-i})
\end{aligned}
$$

Put $i = k$

$$
\begin{aligned}
&= 7^k\,T(1) \\
&= 7^k
\end{aligned}
$$

$$
\begin{aligned}
\text{That is, } T(n) &= 7^{\log_2 n} \\
&= n^{\log_2 7} \\
&= O(n^{\log_2 7}) = O(n^{2.81})
\end{aligned}
$$

So, concluding that Strassen's algorithm is asymptotically more efficient than the standard algorithm. In practice, the overhead of managing the many small matrices does not pay off until 'n' revolves the hundreds.

Quick Sort

The main reason for the slowness of Algorithms like SIS is that all comparisons and exchanges between keys in a sequence $w_1, w_2, \ldots, w_n$ take place between adjacent pairs. In this way it takes a relatively long time for a key that is badly out of place to work its way into its proper position in the sorted sequence.

Hoare his devised a very efficient way of implementing this idea in the early 1960's that improves the $O(n^2)$ behavior of SIS algorithm with an expected performance that is $O(n \log n)$.

In essence, the quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value.

The chosen value is known as the *pivot element*. Once the array has been rearranged in this way with respect to the pivot, the very same partitioning is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition() makes use of two pointers 'i' and 'j' which are moved toward each other in the following fashion:

- Repeatedly increase the pointer 'i' until a[i] >= pivot.
- Repeatedly decrease the pointer 'j' until a[j] <= pivot.

- If j > i, interchange a[j] with a[i]
- Repeat the steps 1, 2 and 3 till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and place pivot element in 'j' pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array 'a' between positions 'low' and 'high'.

- It terminates when the condition low >= high is satisfied. This condition will be satisfied only when the array is completely sorted.
- Here we choose the first element as the 'pivot'. So, pivot = x[low]. Now it calls the partition function to find the proper position j of the element x[low] i.e. pivot. Then we will have two sub-arrays x[low], x[low+1], ... x[j-1] and x[j+1], x[j+2], . . .x[high].
- It calls itself recursively to sort the left sub-array x[low], x[low+1], ... x[j-1] between positions low and j-1 (where j is returned by the partition function).
- It calls itself recursively to sort the right sub-array x[j+1], x[j+2], ... x[high] between positions j+1 and high.

Algorithm Algorithm

```
QUICKSORT(low, high)
/* sorts the elements a(low), . . . . . , a(high) which reside in the global array   A(1 :
n) into ascending order a (n + 1) is considered to be defined and must be greater
than all elements in a(1 : n); A(n + 1) = + ∝ */
{
        if low < high then
        {
                j := PARTITION(a, low, high+1);
                                        // J is the position of the partitioning element
                QUICKSORT(low, j – 1);
                QUICKSORT(j + 1 , high);
        }
}
```

```
Algorithm PARTITION(a, m, p)
{
        V ← a(m); i ← m; j ← p;                    // A (m) is the partition element
        do
        {
                loop  i  := i  + 1  until  a(i) ≥ v          // i moves left to right
                loop  j  := j  - 1  until  a(j) ≤ v          // p moves right to left
                if (i < j) then INTERCHANGE(a, i, j)
         } while (i ≥ j);
        a[m]  :=a[j]; a[j]  :=V;  // the partition element belongs at position P
        return j;
}
```

```
Algorithm INTERCHANGE(a, i, j)
{
        P:=a[i];
        a[i] := a[j];
        a[j] := p;
}
```

Example

Select first element as the pivot element. Move 'i' pointer from left to right in search of an element larger than pivot. Move the 'j' pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped. This process continues till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and interchange pivot and element at 'j' position.

Let us consider the following example with 13 elements to analyze quick sort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*Remarks*
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				i						j			swap i & j
				04						79			
					i			j					swap i & j
					02			57					
						j	i						
(24	08	16	06	04	02)	**38**	(56	57	58	79	70	45)	swap pivot & j
pivot					j, i								swap pivot & j
(02	08	16	06	04)	**24**								
pivot, j	i												swap pivot & j
02	(08	16	06	04)									
	pivot	i		j									swap i & j
		04		16									
			j	i									
	(06	04)	**08**	(16)									swap pivot & j
	pivot, j	i											
	(04)	**06**											swap pivot & j
	04 pivot, j, i												
				16 pivot, j, i									
(02	**04**	**06**	**08**	**16**	**24)**	38							
							(56	57	58	79	70	45)	

							pivot	i				j	swap i & j
								45				57	
								j	i				
							(45)	**56**	(58	79	70	57)	swap pivot & j
							45 pivot, j, i						swap pivot & j
									(58 pivot	79 i	70	57) j	swap i & j
										57		79	
										j	i		
									(57)	**58**	(70	79)	swap pivot & j
									57 pivot, j, i				
											(70	79)	
											pivot, j	i	swap pivot & j
											70		
												79 pivot, j, i	
							(45	**56**	**57**	**58**	**70**	**79)**	
02	**04**	**06**	**08**	**16**	**24**	**38**	**45**	**56**	**57**	**58**	**70**	**79**	

Analysis of Quick Sort:

Like merge sort, quick sort is recursive, and hence its analysis requires solving a recurrence formula. We will do the analysis for a quick sort, assuming a random pivot (and no cut off for small files).

We will take $T(0) = T(1) = 1$, as in merge sort.

The running time of quick sort is equal to the running time of the two recursive calls plus the linear time spent in the partition (The pivot selection takes only constant time). This gives the basic quick sort relation:

$$T(n) = T(i) + T(n - i - 1) + Cn \qquad - \qquad (1)$$

Where, $i = |S_1|$ is the number of elements in S_1.

Worst Case Analysis

The pivot is the smallest element, all the time. Then $i=0$ and if we ignore $T(0)=1$, which is insignificant, the recurrence is:

$$T(n) = T(n - 1) + Cn \qquad n > 1 \qquad - \qquad (2)$$

Using equation – (1) repeatedly, thus

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

$$T(n-2) = T(n-3) + C(n-2)$$

- - - - - - - -

$$T(2) = T(1) + C(2)$$

Adding up all these equations yields

$$T(n) = T(1) + \sum_{i=2}^{n} i$$

$$= \mathbf{O(n^2)} \qquad - \qquad (3)$$

Best Case Analysis

In the best case, the pivot is in the middle. To simply the math, we assume that th two sub-files are each exactly half the size of the original and although this gives slight over estimate, this is acceptable because we are only interested in a Big – o answer.

$$T(n) = 2T(n/2) + Cn \qquad - \qquad (4)$$

Divide both sides by n

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + C \qquad - \qquad (5)$$

Substitute n/2 for 'n' in equation (5)

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + C \qquad - \qquad (6)$$

Substitute n/4 for 'n' in equation (6)

$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + C \qquad - \qquad (7)$$

- - - - - - - -

- - - - - - - -

Continuing in this manner, we obtain:

$$\frac{T(2)}{2} = \frac{T(1)}{1} + C \qquad - \qquad (8)$$

We add all the equations from 4 to 8 and note that there are log n of them:

$$\frac{T(n)}{n} = \frac{T(1)}{1} + C \log n \qquad - \qquad (9)$$

Which yields, $T(n) = C n \log n + n = \mathbf{O(n \log n)}$ - (10)

This is exactly the same analysis as merge sort, hence we get the same answer.

Average Case Analysis

The number of comparisons for first call on partition: Assume left_to_right moves over k smaller element and thus k comparisons. So when right_to_left crosses left_to_right it has made n-k+1 comparisons. So, first call on partition makes n+1 comparisons. The average case complexity of quicksort is

T(n) = comparisons for first call on quicksort
+
{Σ 1<=nleft,nright<=n [T(nleft) + T(nright)]}n = (n+1) + 2 [T(0) +T(1) + T(2) + ----- + T(n-1)]/n

nT(n) = n(n+1) + 2 [T(0) +T(1) + T(2) + ----- + T(n-2) + T(n-1)]

(n-1)T(n-1) = (n-1)n + 2 [T(0) +T(1) + T(2) + ----- + T(n-2)] \

Subtracting both sides:

nT(n) -(n-1)T(n-1) = [n(n+1) - (n-1)n] + 2T(n-1) = 2n + 2T(n-1)
nT(n) = 2n + (n-1)T(n-1) + 2T(n-1) = 2n + (n+1)T(n-1)
T(n) = 2 + (n+1)T(n-1)/n
The recurrence relation obtained is:
T(n)/(n+1) = 2/(n+1) + T(n-1)/n

Using the method of subsititution:

T(n)/(n+1) = 2/(n+1) + T(n-1)/n
T(n-1)/n = 2/n + T(n-2)/(n-1)
T(n-2)/(n-1) = 2/(n-1) + T(n-3)/(n-2)
T(n-3)/(n-2) = 2/(n-2) + T(n-4)/(n-3)
. .
. .
T(3)/4 = 2/4 + T(2)/3
T(2)/3 = 2/3 + T(1)/2 T(1)/2 = 2/2 + T(0)
Adding both sides:
T(n)/(n+1) + [T(n-1)/n + T(n-2)/(n-1) + ------------- + T(2)/3 + T(1)/2]
= [T(n-1)/n + T(n-2)/(n-1) + ------------- + T(2)/3 + T(1)/2] + T(0) +
[2/(n+1) + 2/n + 2/(n-1) + ---------- +2/4 + 2/3]
Cancelling the common terms:
T(n)/(n+1) = 2[1/2 +1/3 +1/4+--------------+1/n+1/(n+1)]

$$T(n) = (n+1)2[\sum_{2\le k\le n+1} 1/k$$

=2(n+1) [-]
=2(n+1)[log (n+1) – log 2]
=2n log (n+1) + log (n+1)-2n log 2 –log 2
T(n)= O(n log n)

3.8. Straight insertion sort:

Straight insertion sort is used to create a sorted list (initially list is empty) and at each iteration the top number on the sorted list is removed and put into its proper place in the sorted list. This is done by moving along the sorted list, from the smallest to the largest number, until the correct place for the new number is located i.e. until all sorted numbers with smaller values comes before it and all those with larger values comes after it. For example, let us consider the following 8 elements for sorting:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Elements	27	412	71	81	59	14	273	87

Solution:

```
Iteration 0:  unsorted   412   71    81    59    14    273   87
              Sorted     27

Iteration 1:  unsorted   412   71    81    59    14    273   87
              Sorted     27    412

Iteration 2:  unsorted   71    81    59    14    273   87
              Sorted     27    71    412

Iteration 3:  unsorted   81    39    14    273   87
              Sorted     27    71    81    412

Iteration 4:  unsorted   59    14    273   87
              Sorted     274   59    71    81    412

Iteration 5:  unsorted   14    273   87
              Sorted     14    27    59    71    81    412

Iteration 6:  unsorted   273   87
              Sorted     14    27    59    71    81    273   412

Iteration 7:  unsorted   87
              Sorted     14    27    59    71    81    87    273   412
```