SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF INFORMATION TECHNOLOGY

UNIT - III

Design and Analysis of Algorithm – SCSA1403

Brute Force And Divide-And-Conquer 9 Hrs.

Brute Force:- Travelling Salesman Problem - Knapsack Problem - Assignment Problem - Closest Pair and Convex Hull Problems - Divide and Conquer Approach:- Binary Search - Quick Sort - Merge Sort - Strassen's Matrix Multiplication.

Brute Force Algorithms

It is a straight forward approach which depends on problem statement and definition. Following algorithms belong to this category."Force" comes from using computer power not intellectual power

Examples

1. Selection Sort
2. Computing a^n ($a > 0$, n a nonnegative integer)
3. Graph Traversal
4. Computing n!
5. Simple Computational Tasks
6. Exhaustive Search
7. Multiplying two matrices
8. Searching for a key of a given value in a list

Strengths

1. Most of the practical problems apply this approach
2. Simple
3. Results in acceptable algorithms for some important problems like matrix multiplication, sorting, searching and string matching

Weaknesses

1. Algorithms cannot be guaranteed as efficient
2. Some of these algorithms are very slow
3. Useful only for instances of small size
4. Not as constructive as some other design techniques

Example 1:

Computing a^n ($a > 0$, n a nonnegative integer) based on the definition of exponentiation

$$a^n = a * a * a * \ldots * a$$

The brute force algorithm requires **n-1** multiplications.

The recursive algorithm for the same problem, based on the observation that $a^n = a^{n/2} * a^{n/2}$ requires **Θ(log (n))** operations.

Travelling Salesman Problem

A complete graph K_N is a graph with N vertices and an edge between every two vertices. Using Hamilton circuit we can find a solution. It is a circuit that uses every vertex of a graph once.

A weighted graph is a graph in which each edge is assigned a weight (representing the time, distance, or cost of traversing that edge).

The Travelling Salesman Problem (TSP) is the problem of finding a minimum-weight Hamilton circuit in K_N

Example:

Given *n* cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city.

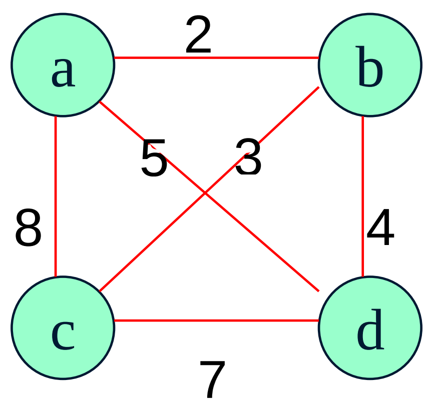
To solve TSP using Brute-force method we can use the following steps:

Step 1. Calculate the total number of tours

Step 2. Draw and list all the possible tours

Step 3. Calculate the distance of each tour

Step 4. Choose the shortest tour, this is the optimal solution



Solution to TSP byExhaustive approach

Tour	**Cost**
a→b→c→d→a	2+3+7+5 = 17
a→b→d→c→a	2+4+7+8 = 21

a→c→b→d→a	8+3+4+5 = 20
a→c→d→b→a	8+7+4+2 = 21
a→d→b→c→a	5+4+3+8 = 20
a→d→c→b→a	5+7+3+2 = 17

Efficiency:Θ((n-1)!)

Knapsack Problem

Given some items, pack the knapsack to get the maximum total value. Each item has some weight and some value. Total weight that we can carry is no more than some fixed number W. So we must consider weights of items as well as their values.

1. Given a knapsack with maximum capacity *W*, and a set *S* consisting of *n* items
2. Each item *i* has some weight w_i and benefit value v_i(all $w_i$and *W* are integer values)
3. Problem: How to pack the knapsack to achieve maximum total value of packed items?

Problem, in other words, is to find

$$\max \sum_{i \in T} v_i \text{ subject to } \sum_{i \in T} w_i \le W$$

Given *n* items:

- weights: $w_1\ w_2\ \ldots\ w_n$
- values: $v_1\ v_2\ \ldots\ v_n$
- a knapsack of capacity *W*

Find most valuable subset of the items that fit into the knapsack

Example: Knapsack capacity W=16

Item	Weight	Value
1	2	$20
2	5	$30
3	10	$50
4	5	$10

Subset	Total weight	Total value
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	15	$80

{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	17	not feasible
{1,2,4}	12	$60
{1,3,4}	17	not feasible
{2,3,4}	20	not feasible
{1,2,3,4}	22	not feasible

Efficiency: $\Theta(2^n)$

Assignment Problem

Let us consider that there are n people and n jobs. Each person has to be assigned only one job. When the j^{th}job is assigned to p^{th}person the cost incurred is represented by C.

$$C=C[p,j]$$

Where, p=1,2,3......n

J=1,2,3.......n

The number of permutations(the number of different assignments to different persons) is n!

The exhaustive search is impractical for large value of n.

Let us consider 4 persons(P1,P2,P3 and P4) and 4 jobs(J1,J2,J3 and J4).

Here n=4.

Here the number of possible and different types of assignment is 4!

n! = 4!

=4 x 3 x 2 x 1

=24

The below table shows the entries representing the assignment costs C[p,j].

Job / Person	J1	J2	J3	J4
P 1	9	2	7	8
P2	6	4	3	7
P3	5	8	1	8
P4	7	6	9	4

Iterations of solving the above assignment problem are given below. Here 4 persons indicated by P1,P2,P3 and P4; Similarly 4 jobs are indicated by J1,J2,J3 and J4.

Let us consider that the assignments can be grouped into 4 groups.

In the first group J1 is assigned to person P1. The remaining jobs J2, J3 and J4 are assigned to persons P2, P3 and P4. The number of ways in which these three jobs can be assigned to three persons is 3!(3!=6).

Group-I

P1	P2	P3	P4	
J1	J2	J3	J4	9+4+1+8 = 18

P1	P2	P3	P4	
J1	J2	J4	J3	9+4+8+9 = 30

P1	P2	P3	P4	
J1	J3	J2	J4	9+3+8+4= 24

P1	P2	P3	P4	
J1	J3	J4	J2	9+3+8+6= 26

P1	P2	P3	P4	
J1	J4	J2	J3	9+7+8+9= 33

P1	P2	P3	P4	
J1	J4	J3	J2	9+7+1+6= 23

Group-2

In the second group J2 is assigned to person P1. The remaining jobs J3,J4,J1 are assigned to persons P2,P3 and P4. The number of ways in which these three jobs can be assigned to three persons is 3!(3!=6).

P1	P2	P3	P4
J2	J3	J4	J1

2+3+8+7 = 20

P1	P2	P3	P4
J2	J3	J1	J4

2+3+5+4 = 14

P1	P2	P3	P4
J2	J4	J3	J1

2+7+1+7= 17

P1	P2	P3	P4
J2	J4	J1	J3

2+7+5+9= 23

P1	P2	P3	P4
J2	J1	J3	J4

2+6+1+4= 13

P1	P2	P3	P4
J2	J1	J4	J3

2+6+8+9= 25

Group-3

In the third group J3 is assigned to person P1. The remaining jobs J2,J4,J1 are assigned to persons P2,P3 and P4. The number of ways in which these three jobs can be assigned to three persons is 3!(3!=6).

P1	P2	P3	P4
J3	J4	J1	J2

7+7+5+6 = 25

P1	P2	P3	P4
J3	J4	J2	J1

7+7+8+7 = 29

P1	P2	P3	P4
J3	J1	J4	J2

7+6+8+6= 27

P1	P2	P3	P4
J3	J2	J4	J1

7+4+8+7= 26

P1	P2	P3	P4
J3	J1	J2	J4

7+6+8+4= 25

P1	P2	P3	P4
J3	J2	J1	J4

7+4+5+4= 20

Group-4

In the Fourth group J4 is assigned to person P1. The remaining jobs J2,J3,J1 are assigned to persons P2,P3 and P4. The number of ways in which these three jobs can be assigned to three persons is 3!(3!=6).

P1	P2	P3	P4
J4	J1	J2	J3

8+6+8+9 = 31

P1	P2	P3	P4
J4	J1	J3	J2

8+6+1+6 = 21

P1	P2	P3	P4
J4	J2	J1	J3

8+4+5+9= 26

P1	P2	P3	P4
J4	J2	J3	J1

8+4+1+7= 20

P1	P2	P3	P4
J4	J3	J1	J2

8+3+5+6= 22

P1	P2	P3	P4
J3	J3	J2	J1

8+3+8+7= 26

In the above four groups low costs are:

Group 1- 1st iteration is lowest	18
Group-II – 5th iteration is lowest	13
Group-III- 6th iteration is lowest	20
Group-IV- 4th iteration is lowest	20

Efficiency – O(n)!

Closest Pair Algorithm

Given n points in the plane, find a pair with smallest Euclidean distance between them. When brute force method is used, it is required to check all pairs of points p and q with $\Theta(n^2)$ comparisons.

Euclidean distance $d(P_i, P_j) =$ Sqrt$[(x_i\text{-}x_j)^2 + (y_i\text{-}y_j)^2]$

Find the minimal distance between a pairs in a set of points

Algorithm BruteForceClosestPoints(P)

```
// P is list of points
dmin ← ∞
for i ← 1 to n-1 do
    for j ← i+1 to n do
        d ← sqrt((xi-xj)^2 + (yi-yj)^2)
        if d<dmin then
            dmin ← d; index1 ← i; index2 ← j
return index1, index2
```

Analysis:

Note the algorithm does not have to calculate the square root

Then the basic operation is squaring

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2$$

$$= 2\sum_{j=i+1}^{n} (n\text{-}i)$$

$$= 2n(n\text{-}1)/2$$

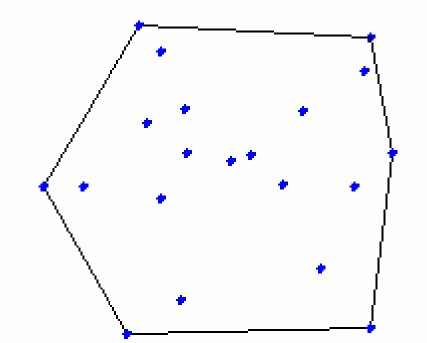
$$\Theta(n^2)$$

Convex Hull Problems

In this problem, we want to compute the convex hull of a set of points?

· Formally: It is the smallest convex set containing the points. A convex set is one in which if we connect any two points in the set, the line segment connecting these points must also be in the set.

· Informally: It is a rubber band wrapped around the "outside" points.



Theorem: The convex hull of any set S of n>2 points (not all collinear) is a convex polygon with the vertices at some of the points of S.

How could you write a brute-force algorithm to find the convex hull?

In addition to the theorem, also note that a line segment connecting two points P1 and P2 is a part of the convex hull's boundary if and only if all the other points in the set lie on thesame side of the line drawn through these points. With a little geometry:

For all points above the line, $ax + by > c$, while for all points below the line, $ax + by < c$. Using these formulas, we can determine if two points are on the boundary to the convex hull.

Algorithm

```
for all points p in S
    for all point q in S
    if p!=q
        Draw a line from p to q
        If all points in S except p and q lie to the left of the line.
Add the directed vector pq to the solution set
```

Efficiency:

$O(n^3)$

Divide and Conquer Algorithm

The divide and conquer methodology is very similar to the modularization approach to software design. Small instances of problem are solved using some direct approach. To solve a large instance, we first divide it into two or smaller instances solve each of these smaller problems and combine the solutions of these smaller problems to obtain the solution to the original instance. The smaller instances are often instances of the original problem and may be solved using divide and conquer strategy recursively.

In Divide and Conquer 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 DivideandConquer (P)
{
if small(P)
then return S(P)
Else
{
divide P into smaller instances P1 ,P2 .....Pk
Apply Divide  and Conquer to each sub problem
Return combine (D and C(P1)+ D and C(P2)+.......+D and C(Pk))
}
}
```

Efficiency Analysis of Divide and Conquer

Let a recurrence relation is expressed as

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

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

Assume $n=b^k$,

$T(b^k)= aT(b^k/b)+f(b^k)$

$T(b^k)= aT(b^{k-1})+f(b^k)$ ....(1)

Assume $n=b^{k-1}$,

$T(b^{k-1})= aT(b^{k-1}/b)+f(b^{k-1})$

$T(b^{k-1})= aT(b^{k-2})+f(b^{k-2})$

Substitute in (1) equation

$T(b^k)= a(aT(b^{k-2})+f(b^{k-2}))+ f(b^k)$

$T(b^k)= a^2T(b^{k-2})+af(b^{k-2})+ f(b^k)$ ....(2)

Assume $n=b^{k-2}$,

$T(b^{k-2})= aT(b^{k-2}/b)+f(b^{k-2})$

$T(b^{k-2})= aT(b^{k-3})+f(b^{k-2})$

Substitute in (2) equation

$T(b^k)= a^2(aT(b^{k-3})+ f(b^{k-2}))+af(b^{k-2})+ f(b^k)$

$T(b^k)= a^3T(b^{k-3})+ a^2f(b^{k-2}))+af(b^{k-2})+ f(b^k)$............(3)

Continuing in this way, we will get

$= a^kT(b^{k-k})+ a^{k-1}f(b^{k-(k-1)}))+ a^{k-2}f(b^{k-(k-2)})) +.........+ af(b^{k-1})+ f(b^k)$

$= a^kT(b^0)+ a^{k-1}f(b^1))+ a^{k-2}f(b^2)) +.........+ af(b^{k-1})+ f(b^k)$

$= a^kT(1)+ a^{k-1}f(b^1))+ a^{k-2}f(b^2)) +.........+ af(b^{k-1})+ f(b^k)$

$= a^kT(1)+ a^{k-1}f(b^1))+ a^{k-2}f(b^2)) +.........+ af(b^{k-1})+ f(b^k)$

$=a^kT(1)+\frac{a^k}{a}+f(b^1)+\frac{a^k}{a^2}f(b^2)+\frac{a^k}{a^3}f(b^3)+\cdots+\frac{a^k}{a^{k-1}}f(b^{k-1})+\frac{a^k}{a^k}f(b^k)$

$=a^k[T(1)+\frac{f(b)}{a}+\frac{f(b^2)}{a^2}+\cdots+\frac{f(b^{k-1})}{a^{k-1}}+\frac{f(b^k)}{a^k}]$

$T(b^k) = a^k[T(1)+\sum_{j=1}^{k}\frac{f(b^j)}{a^j}]$

By property of logarithm,

$a^{log_b x}=x^{log_b a}$

$a^k=a^{log_b n}$

$a^k=n^{log_b a}$

$K=\log_b n$

Substituting the values of a^k and k

$T(b) = a^{\log_b a}[T(1)+\sum_{j=1}^{\log_b n}\frac{f(b^j)}{a^j}$

Binary Search

Binary search method is very fast and efficient. This method requires that the list of elements be in sorted order. Binary search cannot be applied on an unsorted list.

Principle: The data item to be searched is compared with the approximate middle entry of the list. If it matches with the middle entry, then the position will be displayed. If the data item to be searched is lesser than the middle entry, then it is compared with the middle entry of the first half of the list and procedure is repeated on the first half until the required item is found. If the data item is greater than the middle entry, then it is compared with the middle entry of the second half of the list and procedure is repeated on the second half until the required item is found. This process continues until the desired number is found or the search interval becomes empty.

Algorithm:

ALGORITHM BINARYSEARCH(K, N, X)

// K is the array containing the list of data items

// N is the number of data items in the list

// X is the data item to be searched

```
Lower ← 0, Upper ← N – 1
While Lower ≤ Upper
    Mid ←( Lower + Upper ) / 2
    If (X <K[Mid])Then
            Upper ← Mid -1
        Else If (X>K[Mid]) Then
            Lower ← Mid + 1
        Else
            Write("ELEMENT FOUND AT", MID)
            Quit
        End If
    End If
End While
    Write("ELEMENT NOT PRESENT IN THE COLLECTION")
End BINARYSEARCH
```

In Binary Search algorithm given above, K is the list of data items containing N data items. X is the data item, which is to be searched in K. If the data item to be searched is found then the position where it is found will be printed. If the data item to be searched is not found then "Element Not Found" message will be printed, which will indicate the user, that the data item is not found.

Initially lower is assumed 0 to point the first element in the list and upper is assumed as N-1 to point the last element in the list because the range of any array is 0 to N-1. The mid position of the list is calculated by finding the average between lower and upper and X is compared with K[mid]. If X is found equal to K[mid] then the value mid will gets printed, the control comes out of the loop and the procedure comes to an end. If X is found lesser than K[mid], then upper is assigned mid – 1, to search only in the first half of the list. If X is found greater than K[mid], then lower is assigned mid + 1, to search only in the second half of the list. This process is continued until the element searched is found or the collection becomes becomes empty.

Example:

X → Number to be searched :**40**

U → Upper

L → Lower=N-1

M→ Mid

i = 0	**i =1**	**i = 2**	**i = 3**	**i = 4**	**i = 5**	**i = 6**	**i = 7**	**i = 8**	**i = 9**
1	**22**	**35**	**40**	**43**	**56**	**75**	**83**	**90**	**98**

L = 0 M = (0+9)/2 =4 U = 9

X<K[4] → U = 4 – 1 = 3

1	**22**	**35**	**40**	**43**	**56**	**75**	**83**	**90**	**98**

L = 0 M = (0+3)/2=1 U = 3

X >K[1] → L = 1 + 1 = 2

1	**22**	**35**	**40**	**43**	**56**	**75**	**83**	**90**	**98**

L, M = 2 U = 3

K > A [2] → L = 2 + 1 = 3

1	**22**	**35**	**40**	**43**	**56**	**75**	**83**	**90**	**98**

L, M, U = 3

K = A[3] → P = 3 : Number found at position 3

The binarysearch( ) function gets the element to be searched in the variable X. Initially lower is assigned 0 and upper is assumed N – 1. The mid position is calculated and if K[mid] is found equal to X, then mid position will gets displayed. If X is less than K[mid] upper is assigned mid – 1 to search only in first half of the list else lower is assigned mid + 1 to search only in the second half of the list. This is process is continued until lower is less than or equal to upper. If the element is not found even after the loop is completed, then the Not Found Message will be displayed to the user indicating that the element is not found.

Advantages:

1. Searches several times faster than the linear search.
2. In each iteration, it reduces the number of elements to be searched from n to n/2.

Disadvantages:

1. Binary search can be applied only on a sorted list.

Analysis of Binary Search

The basic operation in binary search is comparision of search key with the array elements. To analyze efficiency of binary search we must count the number of times the search key gets compared with the array elements.

In the algorithm after one comparision the array f n elements is divided into n/2 sub arrays.

The worst case efficiency is that the algorithm compares all the array elements for searching the desired element. Hence the worst case time complexity is given by

$$C_{worst}(n) = C_{worst}(n/2) + 1 \quad \text{for } n>1$$

Time Required to compare left sub list middle element or right sub list

One Comparision made with the mid value

$C_{worst}(1) = 1$

When the list is divided, the above equations can be written as

$C_{worst}(n)= C_{worst}(\lfloor n/2 \rfloor) +1$ for $n>1$

$C_{worst}(1) = 1.$

When $n=2^k$, we can write.

(Taking log on both sides $\log_2 n = k\log_2 2 = k$

$C_{worst}(n)= C_{worst}(n/2) +1$ as

$C_{worst}(2^k)= C_{worst}(2^k/2) +1$

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

Using backward substitution method, we can substitute

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

Substitute the value of $C_{worst}(2^{k-1})$ in (1) equation

$C_{worst}(2^k)= [C_{worst}(2^{k-2}) +1] +1$

$= C_{worst}(2^{k-2}) +2$ --------------------------(2)

From (2) equation we can understand that

$C_{worst}(2^{k-2})= C_{worst}(2^{k-3}) +1$

Substitute the value of $C_{worst}(2^{k-2})$ in (2) equation

$C_{worst}(2^k)= [C_{worst}(2^{k-3}) +1] +2$

$= C_{worst}(2^{k-3}) +3$

Continuing upto K

$C_{worst}(2^k)= C_{worst}(2^{k-k}) + k$

$= C_{worst}(2^0) + k$

$= C_{worst}(1) + k$ [$C_{worst}(1) = 1$]

$= 1+k$

When $n=2^k$, we can write.

(Taking log on both sides $\log_2 n = k\log_2 2 = k$

$K= \log_2 n$

$C_{worst}(2^k)= 1+ \log_2 n$

$C_{worst}(2^k)=\log_2 n$

The worst case time complexity of binary search is **O(**$\log_2 n$**)**

Average Case

$1+ \log_2 n = C$

For instance if n=2 then

$\log_2 2 = 1$

Then,

$C=1+1=2$

If n=16, then

$1+ \log_2 16 = C$

$1+4=C$

$C=5$

Then we can write as $C_{average}(n)= 1+ \log_2 n$

$C_{average}(n)=\log_2 n$

The average case time is O($\log_2 n$)

Quick sort

Quick sort is a very popular sorting method. The name comes from the fact that, in general, quick sort can sort a list of data elements significantly faster than any of the common sorting algorithms. This algorithm is based on the fact that it is faster and easier to sort two small lists than one larger one. The basic strategy of quick sort is to divide and conquer. Quick sort is also known as ***partition exchange sort.***

The purpose of the quick sort is to move a data item in the correct direction just enough for it to reach its final place in the array. The method, therefore, reduces unnecessary swaps, and moves an item a great distance in one move.

Principle: A pivotal item near the middle of the list is chosen, and then items on either side are moved so that the data items on one side of the pivot element are smaller than the pivot element, whereas those on the other side are larger. The middle or the pivot element is now in its correct position. This procedure is then applied recursively to the 2 parts of the list, on either side of the pivot element, until the whole list is sorted.

Algorithm:

ALGORITHM QUICKSORT(K, Lower, Upper)

```
// K is the array containing the list of data items
// Lower is the lower bound of the array
// Upper is the upper bound of the array
If (Lower < Upper) Then
BEGIN
        I← Lower + 1
        J ← Upper
        Flag←1
        Key←K[Lower]
        While (Flag)
        BEGIN
                While (K[I] <= Key)
                        I ←I + 1
                End While
                While (K[J] > Key)
                        J ←J – 1
                End While
                If (I < J)Then
                        K[I] ↔ K[J]
                        I←I+1
```

J←J-1

Else

Flag←0

End If

End While

K[J] ↔ K[Lower]

QUICKSORT(K, Lower, J – 1)

QUICKSORT(K, J + 1, Upper)

End If

End QUICKSORT

In Quick sort algorithm, *Lower*points to the first element in the list and the *Upper* points to the last element in the list. Now I is made to point to the next location of *Lower* and J is made to point to the *Upper*.K[Lower] is considered as the *pivot* element and at the end of the pass, the correct position of the *pivot* element will be decided. Keep on incrementing I and stop when K[I] > Key. When I stops, start decrementing J and stop when K[J] < Key. Now check if I < J. If so, swap K[I] and K[J] and continue moving I and J in the same way. When I meets J the control comes out of the loop and K[J] and K[Lower] are swapped. Now the element at position J is at correct position and hence split the list into two partitions: (K{Lower] to K[J-1] and K[J+1] to K[Upper]). Apply the Quick sort algorithm recursively on these individual lists. Finally, a sorted list is obtained.

Example:

N = 10 → Number of elements in the list

U → Upper

L → Lower

i = 0	**i =1**	**i = 2**	**i = 3**	**i = 4**	**i = 5**	**i = 6**	**i = 7**	**i = 8**	**i = 9**
42	**23**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**

L=0 I=0 U, J=9

Initially I=L+1 and J=U, Key=K[L]=42 is the pivot element.

42	**23**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**

L=0 I=2 J=7 U=9

K[2] > Key hence I stops at 2. K[7] < Key hence J stops at 7

Since I < J → Swap K[2] and A[7]

42	**23**	**36**	**11**	**65**	**58**	**94**	**74**	**99**	**87**

L=0 J=3 I=4 U=9

K[4] > Key hence I stops at 4. K[3] < Key hence J stops at 3

Since I > J → Swap K[3] and K[0]. Thus 42 go to correct position.

The list is partitioned into two lists as shown. The same process is applied to these lists individually as shown.

← List 1 →← List 2 →

11	**23**	**36**	**42**	**65**	**58**	**94**	**74**	**99**	**87**

L=0, I=1 J,U=2

(applying quicksort to list 1)

11	**23**	**36**	**42**	**65**	**58**	**94**	**74**	**99**	**87**

L=0, I=1 U=2 J=0 Since I>0 K[L] &K[J] gets swapped i.e., K[0] gets swapped with same element because L,J=0

11	**23**	**36**	**42**	**65**	**58**	**94**	**74**	**99**	**87**

L=4 J=5 I=6 U=9

(applying quicksort to list 2)

(after swapping 58 & 65)

11	**23**	**36**	**42**	**58**	**65**	**94**	**74**	**99**	**87**

L=6 I=8 U, J=9

11	**23**	**36**	**42**	**58**	**65**	**94**	**74**	**87**	**99**

L=6 J=8 U, I=9

11	**23**	**36**	**42**	**58**	**65**	**87**	**74**	**94**	**99**

L=6 U, I, J=7

Sorted List:

11	**23**	**36**	**42**	**58**	**65**	**74**	**87**	**94**	**99**

Analysis of Quicksort:

Algorithm quicksort(A,l,h)

If l<h then

P=partition(a,l,h);

Quicksort(A,l,p-1)

Quicksort(a,p+1,h)

End

Algorithm partition(a,l,h)

Pivot=A[h];

I=l;

For j=l to h do

ifA[i]>pivot then

swqp A[i] with A[j]

i=i+1

swap A[i] with a[h]

returni

End

Analysis

Best Case

If the array is always partitioned at the mid , then it brings the best case efficiency of an algorithm.

The recurrence relation for quick sort for obtaining best case time complexity as

C(n)	=	C(n/2)	+	C(n/2)	+	1
`		Time required to left sub		Time required to right sub		Time required for portioning the sub array

C(1)=0

Using Master theorem we can solve the above equation.

We can write the above equation as

$C(n) = 2C(n/2) + 1$

a=2, b=2, d=1

From Master theorem we get $a=b^d$ $2= 2^1$,

Case 2 satisfied,

So we write as , $C(n)=\Theta(n^d \log n)$

$\Theta(n\log n)$

The time complexity of best case quick sort is $\Theta(n\log n)$

<u>**Worst Case**</u>

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

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

$= \frac{n(n+1)}{2}$

$C(n)= \frac{1}{2}n^2$

$C(n)= \Theta(n^2)$

The time complexity of worst case quick sort is $\Theta(n^2)$

Average Case

The recurrence relation for random input array is

C(n)=C(0)+C(n-1)+n

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

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

..

.

.

C(n)=C(n-1)+C(0)+n

The array value of C(n) is the sum of all the above values divided by n

$C_{avg}(n)=\frac{2\{C(0)+C(1)+C(2)+\cdots.....C(n-1)\}+n.n}{n}$

$C_{avg}(n)=\frac{2}{n}\{C(0)+C(1)+C(2)+\cdots.....C(n-1)\}+n$

Multiplying both sides by n we get,

$nC_{avg}(n)= 2\{C(0)+C(1)+C(2)+\cdots.....C(n-1)\}+n^2$

$C_{avg}(n)=2n \ln n \quad =1.38n \log_2 n$

$C_{avg}(0)=0$ and $C_{avg}(1)=0$

Time Complexity of average case quick sort is $\Theta(n \log_2 n)$

Merge Sort

Principle: The given list is divided into two roughly equal parts called the left and the ight subfiles. These subfiles are sorted using the algorithm recursively and then the two ubfiles are merged together to obtain the sorted file.

Given a sequence of N elements K[0],K[1] ….K[N-1], the general idea is to imagine them split into various subtables of size is equal to 1. So each set will have a individually sorted items with it, then the resulting sorted sequences are merged to produce a single sorted sequence of N elements. Thus this sorting method follows Divide and Conquer strategy. The problem gets divided into various subproblems and by providing the solutions to the subproblems the solution for the original problem will be provided.

Algorithm:

ALGORITHM MERGE(K, low, mid, high)

```
// K is the array containing the list of data items
// Low is the lower bound of the collection
//high is the upper bound of the collection
//mid is the upper bound for the first collection
I ← low, J ← mid+1, L ← 0
While (I ≤ mid) and (J ≤ high)
        If (K[I] < K[J]) Then
                Temp[L] ← K[I]
                I ←I + 1
                L ← L+1
        Else
                Temp[L] ← K[J]
                J ←J + 1
                L ←L + 1
        End If
End While
If (I > mid) Then
        While (J ≤ high)
                Temp[L] ← K[J]
                J ←J + 1
                L ←L + 1
        End While
Else
        While (I ≤ mid)
                Temp[L] ← K[I]
                L ←L + 1
                I ←I + 1
        End While
```

```
End If
Repeat for m = 0 to L step 1
        K[Low+m] ← Temp[m]
End Repeat
End MERGE
```

ALGORITHM MERGESORT(A, low, high)

```
// K is the array containing the list of data items
If (low < high) Then
        mid ← (low + high)/2
        MERGESORT(low, mid)
        MERGESORT(mid + 1, high)
        MERGE(low, mid, high)
End If
End MERGESORT
```

The first algorithm MERGE can be applied on two sorted lists to merge them. nitially, the index variable I points to low and J points to mid + 1. K[I] is compared with ([J] and if K[I] found to be lesser than K[J] then K[I] is stored in a temporary array and I s incremented otherwise K[J] is stored in the temporary array and J is incremented. This :omparison is continued till either I crosses mid or J crosses high. If I crosses the mid irst then that implies that all the elements in first list is accommodated in the temporary ırray and hence the remaining elements in the second list can be put into the temporary ırray as it is. If J crosses the high first then the remaining elements of first list is put as it s in the temporary array. After this process we get a single sorted list. Since this method nerges 2 lists at a time, this is called 2-way merge sort.

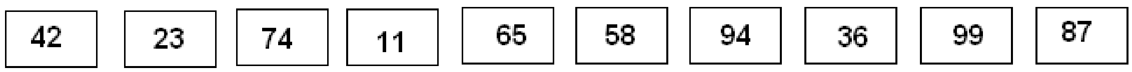
In the MERGESORT algorithm, the given unsorted list is first split into N ıumber of lists, each list consisting of only 1 element. Then the MERGE algorithm is ıpplied for first 2 lists to get a single sorted list. Then the same thing is done on the next wo lists and so on. This process is continued till a single sorted list is obtained.

Example:

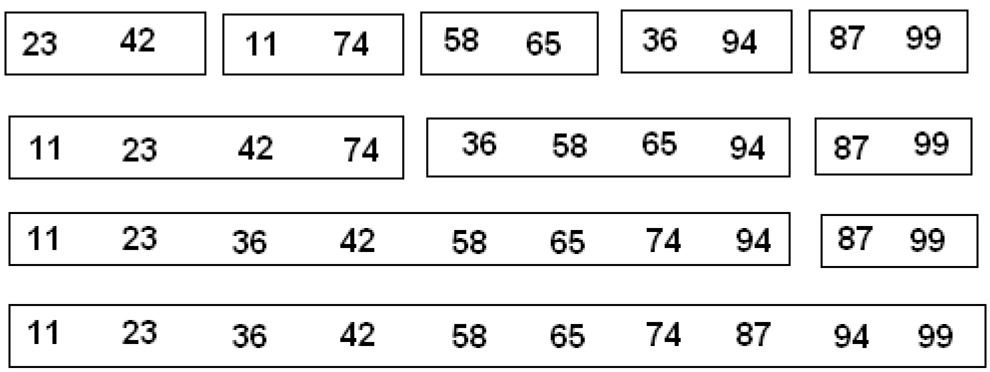
Let L → low, M→ mid, H → high

i = 0	i =1	i = 2	i = 3	i = 4	i = 5	i = 6	i = 7	i = 8	i = 9
42	**23**	**74**	**11**	**65**	**58**	**94**	**36**	**99**	**87**
U				M					H

In each pass the mid value is calculated and based on that the list is split into two. This s done recursively and at last N number of lists each having only one element is roduced as shown.



Now merging operation is called on first two lists to produce a single sorted list, then the ame thing is done on the next two lists and so on. Finally a single sorted list is obtained.



Analysis

First observe that if we call MergeSort with a list containing a single element, then the running time is a constant. Since we are ignoring constant factors, we can just write T (n) =1 . When we call Merge Sort with a list of length n >1 , e.g. Merge(A, low, high), where high −low +1 = n, the algorithm first computes mid= (low+ high) / 2 . The subarray A [low..high] , which contains high −low + 1 elements. You can verify that is of size n/ 2 . Thus the remaining subarray A [mid +1 ..high] has n/ 2 elements in it. How long does it take to sort the left subarray? We do not know this, but because n/ 2< n for n >1 , we can express this as T (n/ 2) . Similarly, we can express the time that it takes to sort the right subarray as T (n/ 2).

Finally, to merge both sorted lists takes n time.

In merge sort algorithms two recursive calls are made.

We can write recurrence relation as

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

T(n/2)	T(n/2)	C(n)
Time taken by left sublist to get sorted	Time taken by right sublist to get sorted	Time taken for combining two sublists

Analysis

Let the recurrence relation for merge sort is

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

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

T(1)=0

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

Apply the Master theorem,

We will get , a=2, b=2, d=1

As per master theorem , $a=b^d$

$T(n)=\Theta(n^d \log_2 n)$

When d=1

$T(n)=\Theta(n\log_2 n)$

The time complexity for merge sort is $\Theta(n\log_2 n)$

Strassen's Matrix Multiplication

The Strassen's method of matrix multiplication is a typical divide and conquer algorithm. With strassens algorithm we can find the product of two 2 by 2 matrices with just seven multiplications. This is obtained by using the following formulas.

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

m1 = (a00 + a11) x (b00 + b11)
m2 = (a10 + a11) x b00
m3 = a00 x (b01 – b11)
m4 = a11 x (b10 – b00)
m5 = (a00 + a01) x b11
m6 = (a10 – a00) x (b00 + b01)
m7 = (a01 – a11) x (b10 + b11)

Example:

$\begin{bmatrix} 3 & 5 \\ 4 & 6 \end{bmatrix} \text{x} \begin{bmatrix} 2 & 7 \\ 8 & 3 \end{bmatrix}$

a00=3, a01= 5, a10=4, a11=6, b00=2, b01=7, b10=8, b11=3

m1 = (a00 + a11) x (b00 + b11)

=(3+6) x (2+3) =9 x 5 = 45

m2 = (a10 + a11) x b00

=(4+6) x 2

=10 x 2 =20

m3 = a00 x (b01 – b11)

=3 x (7-3) = 3 x 4 =12

m4 = a11 x (b10 – b00)

=6 x (8-2) = 6 x 6 =36

m5 = (a00 + a01) x b11

= (3+5) x 3 =24

m6 = (a10 – a00) x (b00 + b01)

=(4-3) x (2+7) =9

m7 = (a01 – a11) x (b10 + b11)

= (5-6) x (8+3)

=(-1) x 11 = -11

m1+m4 - m5+m7 = 45+36-24+(-11) =81-35 =46

m3+m5=12+24 =36

m2+m4=20+36=56

m1+m3-m2+m6=45+12-20+9 =66-20 =46

$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} m1 + m4 - m5 + m7 & m3 + m5 \\ m2 + m4 & m1 + m3 - m2 + m6 \end{bmatrix}$

C= $\begin{bmatrix} 46 & 36 \\ 56 & 46 \end{bmatrix}$

Algorithm:

1. If n = 1 Output A × B
2. Else
3. Compute A00, B01, . . ., A11, B11 % by computing m = n/2
4. m1 ← Strassen(A00, B01 – B11)
5. m2 ← Strassen(A00+ A01, B11)
6. m3 ← Strassen(A10 + A11, B00)
7. m4 ← Strassen(A11, B10 – B00)
8. m5 ← Strassen(A00 + A11, B00 + B11)

9. m6 ← Strassen(A01 – A11, B10 + B11)
10.m7 ← Strassen(A00 − A10, B00 + B01)
11. C 00 ← m5 + m4 − m2 + m6
12. C 01 ← m1 + m2
13. C 10 ← m3 + m4
14. C 11 ← m1 + m5 − m3 − m7
15. Output C
16. End If

Analysis:

The combining cost (lines 12–15) is Θ(n 2) (adding two n/2 × n/2 matrices takes time $n^2/4 = \Theta(n^2)$).

The operations on line 3 take constant time.The combining cost (lines 11–14) is $\Theta(n^2)$. There are 7 recursive calls (lines 4–10). So let T(n) be the total number of mathematical operations performed by Strassen(A, B), then $T(n) = 7T(n^2) + \Theta(n^2)$

The Master Theorem gives us T(n) = Θ(n log2 (7)) = $\Theta(n^{2.8})$.