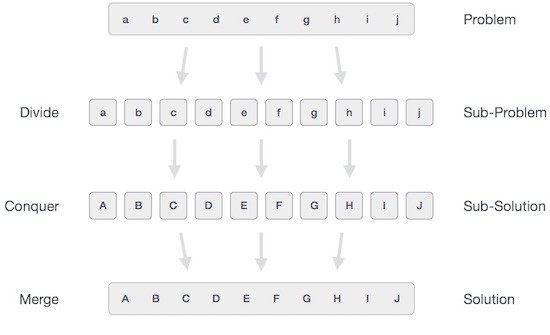
**UNIT II**

**DIVIDE AND CONQUER**

Introduction, Binary Search - Merge sort and its algorithm analysis - Quick sort and its algorithm analysis - Strassen's Matrix multiplication - Finding Maximum and minimum - Algorithm for finding closest pair - Convex Hull Problem

**INTRODUCTION**

In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the sub-problems into even smaller sub-problems, we may eventually reach at a stage where no more division is possible. Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub- problems is finally merged in order to obtain the solution of original problem.



Broadly, we can understand **divide-and-conquer** approach as three step process.

**Divide/Break**

This step involves breaking the problem into smaller sub-problems. Sub-problems should represent as a part of original problem. This step generally takes recursive approach to divide the problem until no sub-problem is further dividable. At this stage, sub-problems become atomic in nature but still represents some part of actual problem.

**Conquer/Solve**

This step receives lot of smaller sub-problem to be solved. Generally at this level, problems

are considered 'solved' on their own.

**Merge/Combine**

When the smaller sub-problems are solved, this stage recursively combines them until they

formulate solution of the original problem.

**Examples**

There are various ways available to solve any computer problem but the following computer

algorithms are based on **divide-and-conquer** programming approach −

Merge Sort

Quick Sort

Binary Search

Strassen's Matrix Multiplication

Closest pair (points)

**BINARY SEARCH ALGORITHM**

Binary search algorithm finds given element in a list of elements with **O(log n)** time complexity where **n** is total number of elements in the list. The binary search algorithm can be used with only sorted list of element. That means, binary search can be used only with list of element which are already arranged in an order. The binary search can not be used for list of element which are in random order. This search process starts comparing of the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for left sublist of the middle element. If the search element is larger, then we repeat the same process for right sublist of the middle element. We repeat this process until we find the search element in the list or until we left with a sublist of only one element. And if that element also doesn't match with the search element, then the result is "Element not found in the list". Binary search is implemented using following steps...

**Step 1:** Read the search element from the user

**Step 2:** Find the middle element in the sorted list

**Step 3:** Compare, the search element with the middle element in the sorted list.

**Step 4:** If both are matching, then display "Given element found!!!" and terminate the function

**Step 5:** If both are not matching, then check whether the search element is smaller or larger

than middle element.

**Step 6:** If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for

the left sublist of the middle element.

**Step 7:** If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for

the right sublist of the middle element.

**Step 8:** Repeat the same process until we find the search element in the list or until sublist

contains only one element.

**Step 9:** If that element also doesn't match with the search element, then display "Element not

found in the list!!!" and terminate the function.

**Pseudo code**

binarysearch(a[n], key, low, high)

while(low<high)

{

mid = (low+high)/2;

if(a[mid]=key)

return mid;

else if (a[mid] > key)

high=mid-1;

else low=mid+1;

}

return -1;

**Program**

#include <stdio.h>

int main()

{

int first, last, middle, size, i, sElement, list[100]; printf("Enter the size of the list: "); scanf("%d",&size);

printf("Enter %d integer values in ascending order\n", size);

for (i = 0; i < size; i++)

scanf("%d",&list[i]);

printf("Enter value to be search: ");

scanf("%d", &sElement);

first = 0;

last = size - 1;

middle = (first+last)/2;

while (first <= last) {

if (list[middle] < sElement)

first = middle + 1;

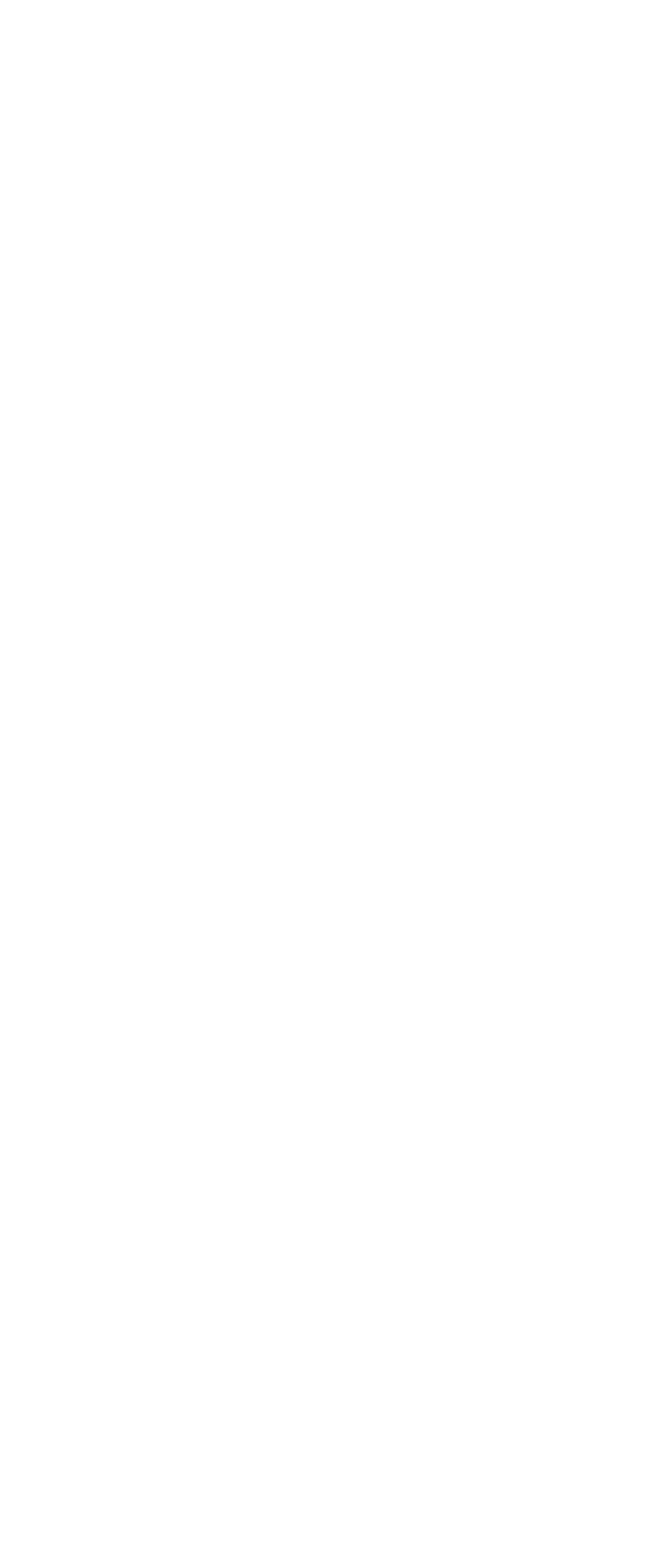
else if (list[middle] == sElement) {

printf("Element found at index %d.\n",middle);

break;

}

else



last = middle - 1;

middle = (first + last)/2;

}

if (first > last)

printf("Element Not found in the list.");

return 0;

}

**Sample input and output:**

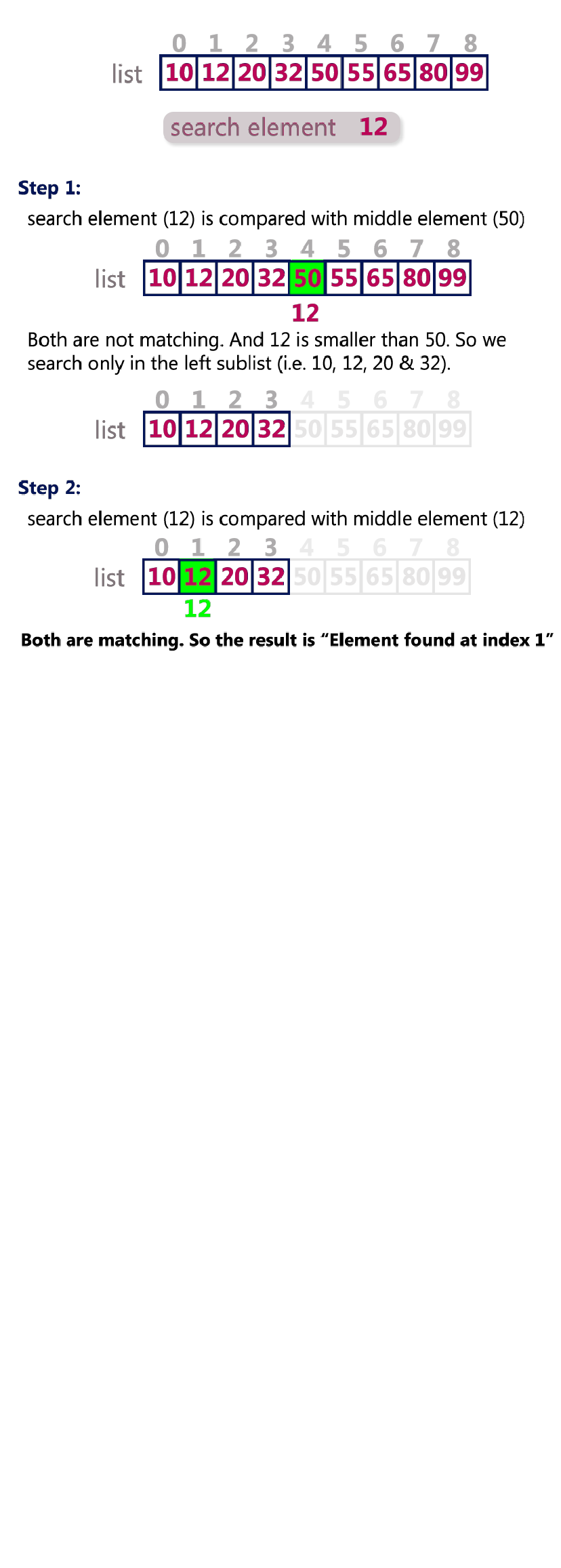
Enter the size of the list:4

Enter 4 integer values in ascending order 4 7 9 11

Enter value to be search: 9

Element found at index 2

**Example:** Consider the following list of elements and search element 12



**15CS204J- ALGORITHM DESIGN AND ANALYSIS**

**Complexity Analysis**

**Worst case analysis:** The key is not in the array

Let T(n) be the number of comparisons done in the worst case for an array of size n. For the purposes of analysis, assume n is a power of

2, i.e. n = 2".

Then #(%) = 2 + #(%/2)

= 2 + 2 + #( + ) // 2nd iteration

-

,

= 2 + 2 + 2 + #(%/2.) // 3rd iteration

...

= / ∗ 2 + #(%/21) // ith iteration

... = 2 ∗ 2 + #(1)

Note that k = logn, and that T(1) = 2. So T(n) = 2logn + 2 = O(logn)

So we expect binary search to be significantly more efficient than linear search for large values of

n.

**MERGE SORT**

Merge sort is based on Divide and conquer method. It takes the list to be sorted and divide it in half to create two unsorted lists. The two unsorted lists are then sorted and merged to get a sorted list. The two unsorted lists are sorted by continually calling the merge-sort algorithm; we eventually get a list of size 1 which is already sorted. The two lists of size 1 are then merged.

**Steps**

1. Input the total number of elements that are there in an array (number\_of\_elements).

2. Input the array (array[number\_of\_elements]).

3. Then call the function MergeSort() to sort the input array. MergeSort() function sorts the array in

the range [left,right] i.e. from index left to index right inclusive.

4. Merge() function merges the two sorted parts. Sorted parts will be from [left, mid] and [mid+1,

right]. After merging output the sorted array.

**Algorithm**

Sort the entire sequence A[1 .. n], make the initial call to the procedure MERGE-SORT (*A*, 1, *n*).

MERGE-SORT (*A*, *p*, *r*)

1. WHILE *p* < *r* // Check for base case

2. *q* = FLOOR[(*p* + *r*)/2] // Divide step

3. MERGE-SORT (A, *p*, *q*)

4. MERGE-SORT (A, *q* + 1, *r*)

5. MERGE (A, *p*, *q+1*, *r*)

**MergeSort() function / Split step**

For *mergesort* function we get three parameters, the input array *a[], the start index and the end* index of array. This start and end change in every recursive invocation of mergesort function. We find the middle index using start and end index of the input array and again invoke the same

function with two parts one starting from start to mid and other being from mid+1 to end. Once base condition is hit, we start winding up and call merge function. *Merge* function takes four parameters, *input array, start, end and middle index* based on start and end.

**Merge() function / Merge step**

Sort the smallest parts and combine them together with merge operation. In merge operation, scan both sorted arrays and based on the comparison, put one of the two items into output array, till both arrays are scanned. If one array is finished before other, just copy all elements from the other array to output array. Copy this output array back to original input array so that it can be combined with bigger sub problems till solution to original problem is derived.

Merge function merges two sub arrays (one from start to mid and other from mid+1 to end) into a single array from start to end. This array is then returned to upper calling function which then again sort two parts of array.

The **pseudocode** of the MERGE procedure is as follow:

MERGE (*A*, *p*, *q*, *r* )

1. *n*1 ← *q* − *p* + 1

2. *n*2 ← *r* − *q*

3. Create arrays L[1 . . *n*1 + 1] and R[1 . . *n*2 + 1]

4. **FOR** *i* ← 1 **TO** *n*1

5. **DO** L[*i*] ← A[*p* + *i* − 1]

6. **FOR** *j* ← 1 **TO** *n*2

7. **DO** R[*j*] ← A[*q* + *j* ]

8. L[*n*1 + 1] ← ∞

9. R[*n*2 + 1] ← ∞

10. *i* ← 1

11. *j* ← 1

12. **FOR** *k* ← *p* **TO** *r*

13. **DO IF** L[*i* ] ≤ R[ *j*]

14. **THEN** A[*k*] ← L[*i*]

15. *i* ← *i* + 1

16. **ELSE** A[k] ← R[j]

17. *j* ← *j* + 1

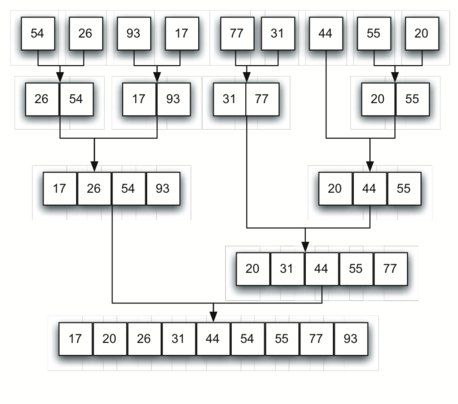
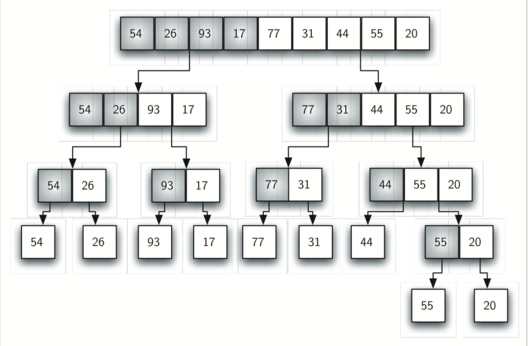
**Properties:**

Best case – When the array is already sorted O(nlogn).

Worst case – When the array is sorted in reverse order O(nlogn). Average case – O(nlogn).

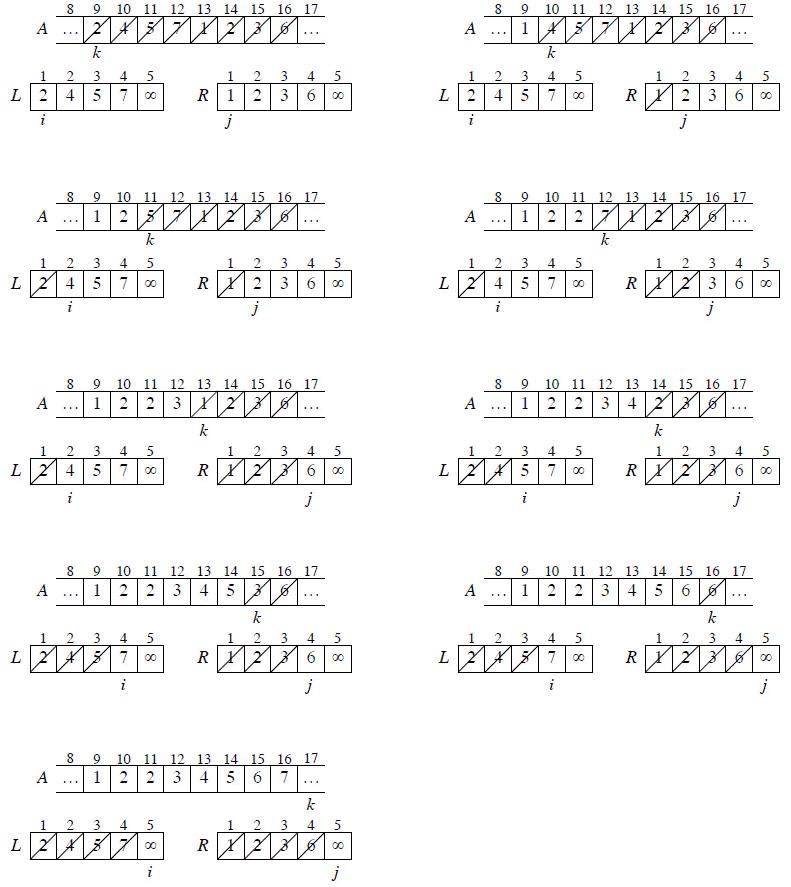
Extra space is required, so space complexity is O(n) for arrays and O(logn) for linked lists.

**Example**



**Example for merge**

**Complexity Analysis**



Every time input space is divided into two, this division will have complexity of O(log n) where n is input size. First part or split part of implementation of merge sort has complexity of O(logn). Now the second part of implements merge step which place every element in its proper place in array, hence it linear time O(n). Since above step of dividing has to be done for n elements, hence total complexity of merge sort will be O(nlogn).

**Analyzing Merge Sort**

For simplicity, assume that *n* is a power of 2 so that each divide step yields two subproblems, both

of size exactly *n*/2.

The base case occurs when *n* = 1. When *n* ≥ 2, time for merge sort steps:

**Divide**: Just compute *q* as the average of *p* and *r*, which takes constant time i.e. Θ(1).

**Conquer**: Recursively solve 2 subproblems, each of size *n*/2, which is 2T(*n*/2).

**Combine**: MERGE on an *n*-element subarray takes Θ(*n*) time.

Summed together they give a function that is linear in *n*, which is Θ(*n*). Therefore, the recurrence

for merge sort running time is



**Solving the Merge Sort Recurrence**

By the master theorem in CLRS-Chapter 4 (page 73), we can show that this recurrence has the

solution

T(*n*) = Θ(*n* lg *n*).

Reminder: lg *n* stands for log2 *n*.

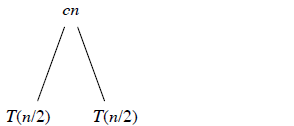
Compared to insertion sort [Θ(*n*2) worst-case time], merge sort is faster. Trading a factor of *n* for a factor of lg *n* is a good deal. On small inputs, insertion sort may be faster. But for large enough inputs, merge sort will always be faster, because its running time grows more slowly than insertion sorts.

**Recursion Tree**

We can understand how to solve the merge-sort recurrence without the master theorem. There is a drawing of recursion tree on page 35 in CLRS, which shows successive expansions of the recurrence.

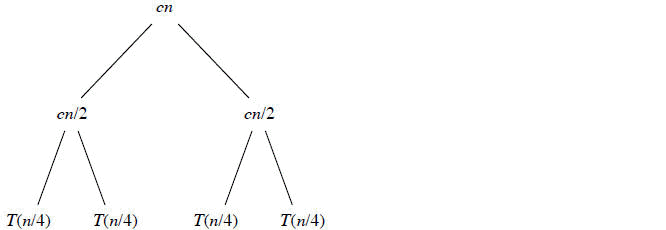
The following figure (Figure 2.5b in CLRS) shows that for the original problem, we have a cost of

*cn*, plus the two subproblems, each costing T (*n*/2).



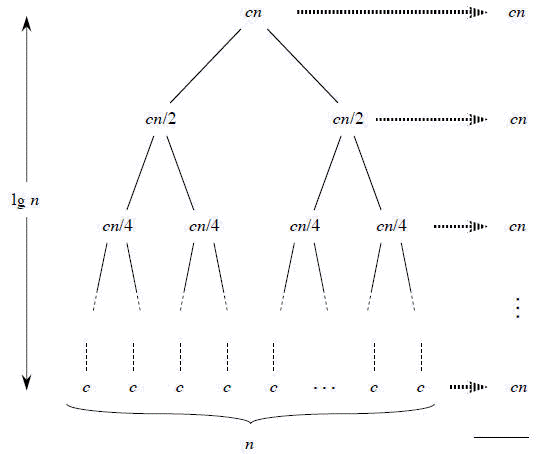
The following figure (Figure 2.5c in CLRS) shows that for each of the size-*n*/2 subproblems, we

have a cost of *cn*/2, plus two subproblems, each costing T (*n*/4).



The following figure (Figure: 2.5d in CLRS) tells to continue expanding until the problem sizes get

down to 1.



In the above recursion tree, each level has cost *cn*.

The top level has cost *cn*.

The next level down has 2 subproblems, each contributing cost *cn*/2.

The next level has 4 subproblems, each contributing cost *cn*/4.

Each time we go down one level, the number of subproblems doubles but the cost per

subproblem halves. Therefore, cost per level stays the same. The height of this recursion tree is lg *n* and there are lg *n* + 1 levels.

**Mathematical Induction**

We use induction on the size of a given subproblem *n*.

**Base case**: *n* = 1

Implies that there is 1 level, and lg 1 + 1 = 0 + 1 = 1.

**Inductive Step**

Our inductive hypothesis is that a tree for a problem size of 2*i* has lg 2*i* + 1 = *i* +1 levels. Because we assume that the problem size is a power of 2, the next problem size up after 2*i* is 2*i* + 1. A tree for a problem size of 2*i* + 1 has one more level than the size-2*i* tree implying *i* + 2 levels.

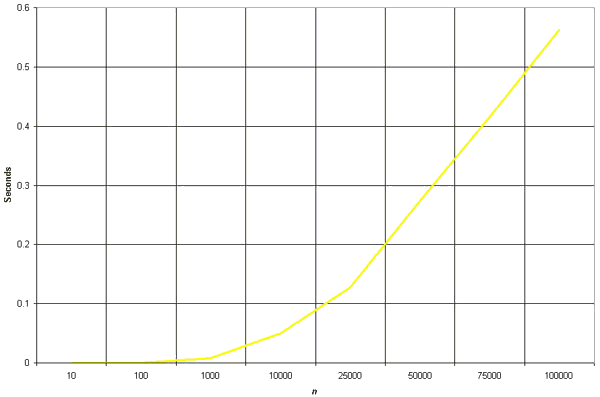
Since lg 2*i* + 1 = *i* + 2, we are done with the inductive argument.

Total cost is sum of costs at each level of the tree. Since we have lg *n* +1 levels, each costing *cn*, the

total cost is *cn* lg n + *cn*.

Ignore low-order term of *cn* and constant coefficient *c*, and we have, Θ(*n* lg *n*) which is the desired

result.



**QUICK SORT AND ITS ALGORITHM ANALYSIS**

Quick sort works by partitioning a given array *A*[*p* . . *r*] into two non-empty sub array *A*[*p* . . *q*] and *A*[*q*+1 . . *r*] such that every key in *A*[*p* . . *q*] is less than or equal to every key in *A*[*q*+1 . . *r*]. Then the two sub arrays are sorted by recursive calls to Quick sort. The exact position of the partition depends on the given array and index q is computed as a part of the partitioning procedure.

**QuickSort ()**

1. If p < r then

2. Partition (A, p, r)

3. Recursive call to Quick Sort (A, p, q)

4. Recursive call to Quick Sort (A, q + r, r)

Note that to sort entire array, the initial call Quick Sort (*A*, 1, length[*A*])

As a first step, Quick Sort chooses as pivot one of the items in the array to be sorted. Then array is then partitioned on either side of the pivot. Elements that are less than or equal to pivot will move toward the left and elements that are greater than or equal to pivot will move toward the right.

**Partitioning the Array**

Partitioning procedure rearranges the sub arrays in-place.

**PARTITION (*A*, *p*, *r*)**

1. x ← A[p]

2. i ← p-1

3. j ← r+1

4. while TRUE do

5. Repeat j ← j-1

6. until A[j] ≤ x

7. Repeat i ← i+1

8. until A[i] ≥ x

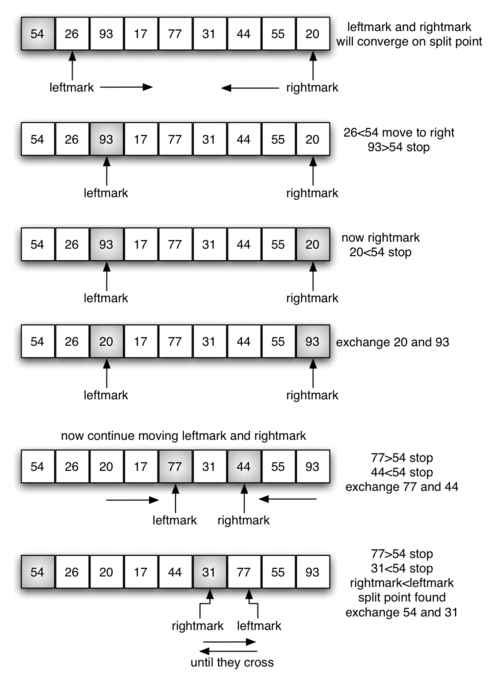
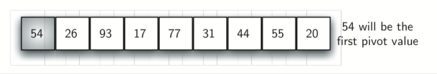
9. if i < j

10. then exchange A[i] ↔ A[j]

11. else return j

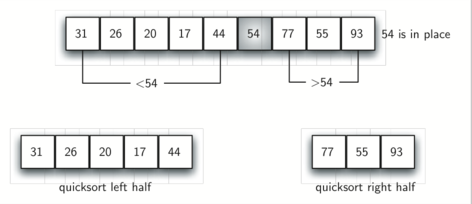
**Example :**

Partitioning begins by locating two position markers—let’s call them leftmark and rightmark— at the beginning and end of the remaining items in the list (positions 1 and 8). The goal of the partition process is to move items that are on the wrong side with respect to the pivot value while also converging on the split point. This process as we locate the position of 54.



We begin by incrementing leftmark until we locate a value that is greater than the pivot value. We then decrement rightmark until we find a value that is less than the pivot value. At this point we have discovered two items that are out of place with respect to the eventual split point. For our example, this occurs at 93 and 20. Now we can exchange these two items and then repeat the process again.

At the point where rightmark becomes less than leftmark, we stop. The position of rightmark is now the split point. The pivot value can be exchanged with the contents of the split point and the pivot value is now in place. In addition, all the items to the left of the split point are less than the pivot value, and all the items to the right of the split point are greater than the pivot value. The list can now be divided at the split point and the quick sort can be invoked recursively on the two halves.



**Implementation in C**

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than specified value say pivot based on which the partition is made and another array holds values greater than pivot value.

#include <stdio.h>

#include <stdbool.h>

#define MAX 7

int intArray[MAX] = {4,6,3,2,1,9,7};

void printline(int count){

int i;

for(i = 0;i <count-1;i++){

printf("=");

}

printf("=\n");

}

Void display(){ int i; printf("[");

// navigate through all items for(i = 0;i<MAX;i++){

printf("%d ",intArray[i]);

}

printf("]\n");

}

void swap(int num1, int num2){ int temp = intArray[num1]; intArray[num1] = intArray[num2]; intArray[num2] = temp;

}

int partition(int left, int right, int pivot){

int leftPointer = left -1;

int rightPointer = right;

while(true){

while(intArray[++leftPointer] < pivot){

//do nothing

}

while(rightPointer > 0 && intArray[--rightPointer] > pivot){

//do nothing

}

if(leftPointer >= rightPointer){

break;

}else{

printf(" item swapped :%d,%d\n", intArray[leftPointer],intArray[rightPointer]); swap(leftPointer,rightPointer);

}

}

printf(" pivot swapped :%d,%d\n", intArray[leftPointer],intArray[right]);

swap(leftPointer,right); printf("Updated Array: "); display();

return leftPointer;

}

void quickSort(int left, int right){

if(right-left <= 0){

return;

}else {

int pivot = intArray[right];

int partitionPoint = partition(left, right, pivot); quickSort(left,partitionPoint-1); quickSort(partitionPoint+1,right);

}

}

main(){

printf("Input Array: ");

display(); printline(50); quickSort(0,MAX-1); printf("Output Array: "); display();

printline(50);

}

If we compile and run the above program then it would produce following result −

**Output**

Input Array: [4 6 3 2 1 9 7 ]

==================================================

pivot swapped :9,7

Updated Array: [4 6 3 2 1 7 9 ]

pivot swapped :4,1

Updated Array: [1 6 3 2 4 7 9 ]

item swapped :6,2 pivot swapped :6,4

Updated Array: [1 2 3 4 6 7 9 ]

pivot swapped :3,3

Updated Array: [1 2 3 4 6 7 9 ] Output Array: [1 2 3 4 6 7 9 ]

==================================================

**Analysis of Quick sort**

**Best Case**

The best thing that could happen in Quick sort would be that each partitioning stage divides the array exactly in half. In other words, the best to be a median of the keys in *A*[*p* . . *r*] every time procedure 'Partition' is called. The procedure 'Partition' always split the array to be sorted into two equal sized arrays.

If the procedure 'Partition' produces two regions of size *n*/2. the recurrence relation is then T(n) = T(*n*/2) + T(*n*/2) + (*n*)



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



And from case 2 of Master theorem

T(n) = (*n* lg *n*)



**Worst-case**

Let *T*(*n*) be the worst-case time for QUICK SORT on input size n. We have a recurrence

*T*(*n*) = max1≤*q*≤*n*-1 (*T*(*q*) + *T*(*n*-*q*)) + (*n*) --------- 1



where *q* runs from 1 to *n*-1, since the partition produces two regions, each having size at least 1.

Now we guess that *T*(*n*) ≤ *cn*2 for some constant *c*. Substituting our guess in equation 1.We get

*T*(*n*) = max1≤*q*≤*n*-1 (*cq*2 ) + *c*(*n* - *q*2)) + (*n*)



= *c* max (*q*2 + (*n* - *q*)2) + (*n*)



Since the second derivative of expression *q*2 + (*n*-*q*)2 with respect to q is positive. Therefore, expression achieves a maximum over the range 1≤ q ≤ n -1 at one of the endpoints. This gives the bound max (*q*2 + (*n* - *q*)2)) 1 + (*n* -1)2 = *n*2 + 2(*n* -1).

Continuing with our bounding of *T*(*n*) we get

*T*(*n*) ≤ *c* [*n*2 - 2(*n*-1)] + (*n*)



= *cn*2 - 2*c*(*n*-1) + (*n*)



Since we can pick the constant so that the 2*c*(*n* -1) term dominates the (*n*) term we have



*T*(*n*) ≤  *cn*2

Thus the worst-case running time of quick sort is (*n2*).



**STRASSEN'S MATRIX MULTIPLICATION**

Given two square matrices A and B of size n x n each, find their multiplication matrix.

***Naive Method***

Following is a simple way to multiply two matrices.

void multiply(int A[][N], int B[][N], int C[][N])

{

for (int i = 0; i < N; i++)

{

for (int j = 0; j < N; j++)

{

C[i][j] = 0;

for (int k = 0; k < N; k++)

{

C[i][j] += A[i][k]\*B[k][j];

}

}

}

}

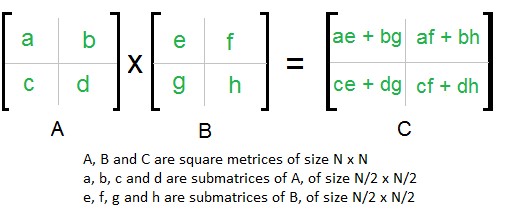
Time Complexity of above method is O(N3).

***Divide and Conquer***

Following is simple Divide and Conquer method to multiply two square matrices.

1) Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.

2) Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.



In the above method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions. Addition of two matrices takes O(N2) time. So the time complexity can be written as

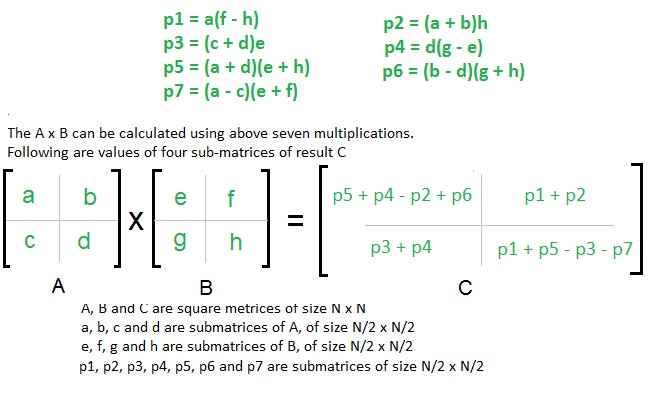
T(N) = 8T(N/2) + O(N2)

From Master's Theorem, time complexity of above method is O(N3) which is unfortunately same

as the above naive method.

***Simple Divide and Conquer also leads to O(N3), can there be a better way?***

In the above divide and conquer method, the main component for high time complexity is 8 recursive calls. The idea of Strassen’s method is to reduce the number of recursive calls to 7. Strassen’s method is similar to above simple divide and conquer method in the sense that this method also divides matrices to sub-matrices of size N/2 x N/2 as shown in the above diagram, but in Strassen’s method, the four sub-matrices of result are calculated using following formulae.



**Time Complexity of Strassen’s Method**

Addition and Subtraction of two matrices takes O(N2) time. So time complexity can be written as T(N) = 7T(N/2) + O(N2).

From Master's Theorem, time complexity of above method is O(NLog7) which is approximately

O(N2.8074)

**Generally, Strassen’s Method is not preferred for practical applications for following**

**reasons.**

1. The constants used in Strassen’s method are high and for a typical application Naive

method works better.

2. For Sparse matrices, there are better methods especially designed for them.

3. The sub matrices in recursion take extra space.

4. Because of the limited precision of computer arithmetic on non integer values, larger errors

accumulate in Strassen’s algorithm than in Naive Method

**FINDING MAXIMUM AND MINIMUM**

**Problem: Finding the maximum and minimum elements in a set of (n) elements using the**

**straightforward algorithm.**

**Algorithm straightforward** (a, n, max, min)

*Input*: array (a) with (n) elements

*Output*: max: max value, min: min value max = min = a(1)

for i = 2 to n do begin

end

end

if (a(i)>max) then max=a(i)

if (a(i)<min) then min=a(i)

**best= average =worst= 2(n-1) comparisons**

If we change the body of the loop as follows:

max=min=a(1)

for i=2 to n do begin

end

**Complexity**

if (a(i)>max) then max=a(i)

else if (a(i)<min) then min=a(i)

best case: elements in increasing order No. of comparisons = (n-1)

Worst case: elements in decreasing order No. of comparisons = 2(n-1)

Average case: a(i) is greater than max half the time No. of comparisons= 3n/2 – 3/2

½ ((n-1)+(2n-2))

**Divide and Conquer approach**

Problem: is to find the maximum and minimum items in a set of (n) elements.

Algorithm MaxMin(i, j, max, min)

*input:* array of N elements, i lower bound, j upper bound

*output:* max: largest value

min: smallest value. if (i=j) then max=min=a(i)

else

if (i=j-1) then

if (a(i)<a(j)) then max= a(j)

min= a(i)

else

else

max= a(i)

min= a(j)

end

mid=(i+j)/2

maxmin(i, mid, max, min)

maxmin(mid+1, j, max1, min1)

if (max<max1) then max = max1 if (min>min1) then min = min1

**Complexity Analysis**

1.if size= 1 return current element as both max and min //base condition

2.else if size= 2 one comparison to determine max and min //base condition

3.else /\* if size > 2 find mid and call recursive function \*/

recur for max and min of left half recur for max and min of right half.

one comparison determines to max of the two subarray, update max.

one comparison determines min of the two subarray, update min.

4. finally return or print the min/max in whole array.

Recurrence relation can be written as T(n)=2(T/2)+2 solving which give you T(n) =3/2n-2 which

is the exact no. of comparisons but still the worst time complexity will be T(n)=*O*(*n*) and best case time complexity will be *O*(1) when you have only one element in array, which will be candidate for both max and min.

*n*

*n*

*C* 2 *C* 2 2

*for n*  2

*Cn*  1

0

*n*  2 *n*  1

*n*  2*k*

When n is a power of 2,

*Cn*  2*Cn*  2

2

2 2*Cn*  2 2

4

4*Cn*  4 2

4

!

2*k* 1

*C*2

*k* 1

*i*

2

*i*

1

2*k* 1 2*k*  2

3*n*

2

2

Note:

No algorithm based on comparison can do less than this.