**UNIT – II**

* Operations on disjoint sets
* Union and find algorithms
* Spanning tree Algorithm
* Introduction to divide and conquer technique,applications
* Binary search and its complexity
* Merge sort and its complexity
* Quick sort and its complexity
* Multiplication of large integers
* Strassen’s matrix multiplication

****

**UNION AND FIND :**

[**2.9**](https://www.geeksforgeeks.org/easy/)

A [*disjoint-set data structure*](http://en.wikipedia.org/wiki/Disjoint-set_data_structure) is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. A [*union-find algorithm*](http://en.wikipedia.org/wiki/Disjoint-set_data_structure) is an algorithm that performs two useful operations on such a data structure:

***Find:***

Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

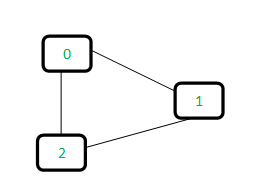
***Union:***

 Join two subsets into a single subset.

*Union-Find Algorithm* is an application of Disjoint Set Data Structure. The application is to check whether a given graph contains a cycle or not.

*Union-Find Algorithm* can be used to check whether an undirected graph contains cycle or not. Note that we have discussed an [algorithm to detect cycle](https://www.geeksforgeeks.org/archives/18516). This is another method based on *Union-Find*. This method assumes that graph doesn’t contain self-loops.  
We can keeps track of the subsets in a 1D array, lets call it parent[].

Let us consider the following graph:

  
For each edge, make subsets using both the vertices of the edge. If both the vertices are in the same subset, a cycle is found.

Initially, all slots of parent array are initialized to -1 (means there is only one item in every subset).

0 1 2

-1 -1 -1

Now process all edges one by one.

*Edge 0-1:*

Find the subsets in which vertices 0 and 1 are. Since they are in different subsets, we take the union of them. For taking the union, either make node 0 as parent of node 1 or vice-versa.

0 1 2 <----- 1 is made parent of 0 (1 is now representative of subset {0, 1})

1 -1 -1

*Edge 1-2:*

1 is in subset 1 and 2 is in subset 2. So, take union.

0 1 2 <----- 2 is made parent of 1 (2 is now representative of subset {0, 1, 2})

1 2 -1

*Edge 0-2:* 0 is in subset 2 and 2 is also in subset 2. Hence, including this edge forms a cycle.

How subset of 0 is same as 2?

0->1->2 // 1 is parent of 0 and 2 is parent of 1

int find(int parent[], int i)

{

if (parent[i] == -1)

return i;

return find(parent, parent[i]);

}

void Union(int parent[], int x, int y)

{

int xset = find(parent, x);

int yset = find(parent, y);

parent[xset] = yset;

}



**Divide and conquer strategy:**

Given a procedure to compute on ‘n’ inputs.The DAC splitting the inputs into k different subsets i.e k different subproblems.

Every DAC strategy involves 3 steps:

i.Divide

ii.Conquer

iii.Combine

* ***Divide:***

Divides the problem into n no. of problems.

* ***Conquer:***

Solve the individual sub problem, if it is smaller otherwise apply the divide.

* ***Combine:***

Combining the solutions of the sub problems to one solution for the original problem.

***Algorithm:***

Algorithm DAC(p)

{

if small(p) then return s(p);

else

{

divide p into smaller instaances p1,p2,…..pk.

apply DAC to each of these problems

return combine(DAC(p1),DAC(p2)…..DAC(pk));

}

}

***Analysis of Divide and Conquer:***

We can compute the runtime of DAC as follows:

STEP 1:

Given problem size is n and if n is small then directly we compute the solution without applying DAC. So that the time required isT(1).

STEP 2:

If the size of ‘p’ is ‘n’ and if it is not smaller then divide the problem into k sub problems n1,n2…..nk.Each of size is 1/b where b is a constant.

Hence the time required to compute the given problem is

T(n) = T(n1/b)+T(n2/b)+……….+T(nk/b)+f(n);

Where f(n) represents the time required to combine the sub-problems.

**Substitution method:**

T(n) = a+(n/b)+f(n);a

a=2,b=2; f(n) = n

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

By substitution method,

T(n) = 2{ 2 T((n/2)/2) +(n/2));

= 4 T(n/4) +n+n;

= 4 T(n/4)+2n;

Again by substitution

= 4 (2T((n/4)/2) +(n/4))

= 8T(n/8) + 3n

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

Let 3=i

Generalised form T(n) = 2^I T(n/2^i) + in

Let n=2^i

* log n = log 2^i
* logn = i log 2
* log n = i

substitute n=2^I; i = logn

T(n) = nT(n/n) + logn^n

= nT(1)+n log n

* **T(n) = O(n logn)**

***Master’s theorem:***

i.f(n) = g(n) => O(f(n)logn)

ii.f(n) < g(n) => O(g(n))

iii.f(n) > g(n) => O(n)

**BINARY SEARCH:**

1)Let ai,1<=i<=n be a list of elements that are sorted in non-desending order.

2)Suppose if we have 'n' nnumber of elements then x is element searched.

3)If n=1,small(p) be true then there is no requirement of applying divide and conqure,if n>1 then it can be divided into new such problems.

4)Pick an index 'q' in the range[i,l] and compare x with aq then the following 3 possibilities will occur,

a)x=aq ,then search element found,

b)x<aq,then is searched in left sublist,

c)x>aq,then x is searched in right sublist.m

**q**

**a0, a1,…………………………..aq-1 aq+1 , ……….an**

0 1 2 3 4 5

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 |

**EXAMPLE:**

Consider the list n=6 where a={1,2,3,4,5,6}

X=4(key element which has to be find)

First find the mid value of the given array

Mid=low+high/2

Low=first element in the array(array index)

High=last element in the array(array index)

Mid=0+5/2

=2.5=2

**Cases:-**

1)compare the key element with a[mid]

X==a[mid]

4==a[2]

4==3(false)

2)check if the key element is bigger or smmaler then mid element

X<=a[mid] X>=a[mid]

4<=a[2] 4>=a[2]

4<3 4>3

\*As said in the procedure key element in bigger than respective array element so the key element will be in right sub list of the array

\*Again find the mid for the right sublist

Now low=mid+1

=2+1=3

\*High will not be changed because we are searching in the right sublist

|  |  |  |
| --- | --- | --- |
| 4  3 | 5  4 | 6  5 |

Mid =3+5/2

=4

a[mid]=a[4]

=5

**Cases:-**

1)compare keyelement with new mid value

X==a[mid] X<=a[mid]

4==5(false) 4<=5(true)

The key element is less than mid value so now we have the caclulate mid again to find key element

Now low=3

High value changes now

high=mid-1

=4-1=3

Mid=low+high/2

=3+3/2

=3

a[mid]=a[3]=4

cases:-

4==a[3]

4==4(true)

ANALYSIS:-

TIME COMPLEXITY

T(n)=T(n/2)+1.it is in the form of T(n)=aT(n/b)+f(n)

Here a=1,b=2,f(n)=1

By using master’s theorem

1)f(n)=g(n) => O(f(n)logn)

2)f(n)<g(n) => O(g(n))

3)f(n)>g(n) => O(n)

g(n)=nlogba

g(n)=nlog21

=n0

=1

f(n)=g(n) => 1=1

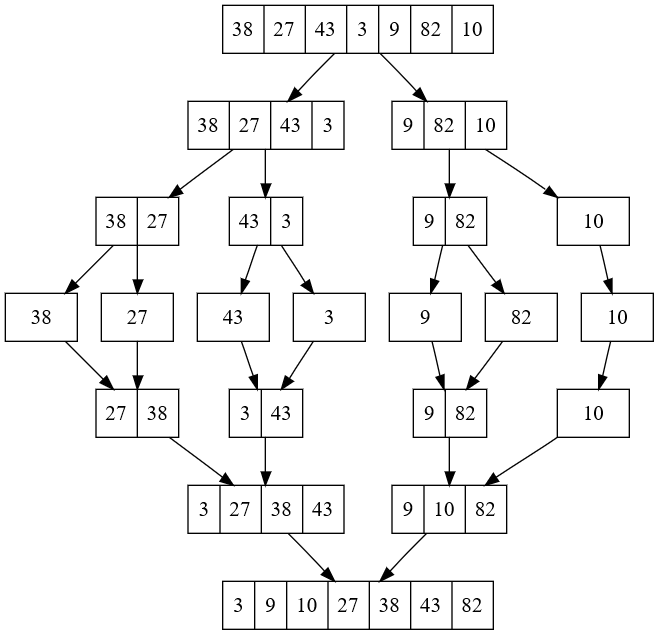
time complexity T(n)=O(f(n)logn)

**=O(logn)**

**MERGE SORT:**

In [computer science](https://en.wikipedia.org/wiki/Computer_science), **merge sort** (also commonly spelled **mergesort**) is an efficient, general-purpose, [comparison-based](https://en.wikipedia.org/wiki/Comparison_sort) [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm). Most implementations produce a [stable sort](https://en.wikipedia.org/wiki/Sorting_algorithm#Stability), which means that the implementation preserves the input order of [equal](https://en.wikipedia.org/wiki/Equality_(mathematics))elements in the sorted output. Mergesort is a [divide and conquer algorithm](https://en.wikipedia.org/wiki/Divide_and_conquer_algorithm) that was invented by [John von Neumann](https://en.wikipedia.org/wiki/John_von_Neumann) in 1945. A detailed description and analysis of bottom-up mergesort appeared in a report by [Goldstine](https://en.wikipedia.org/wiki/Herman_Goldstine) and [von Neumann](https://en.wikipedia.org/wiki/John_von_Neumann) as early as1948.

**EXAMPLE:**



**MERGE SORT ALGORITHM:**

Algorithm mergsort(a,low,high){

If(low<high){

mid:=(low+high)/2;

mergesort(a,low,mid);

mergesort(a,mid+1,high);

merge(a,low,mid,high);

}

}

Algorithm merge(a,low,mid,high){

i:=low;

j:=mid+1;

k:=high;

while(i<=mid and j<=high){

if(a[i]<=a[j]) then{

b[k]:=a[j];

j:=j+1;

k:=k+1;

}

While(i<=mid){

b[k]:=a[j];

i:=i+1;

k:=k+1;

}

While(j<=high){

b[k]:=a[j];

j:=j+1;

k:=k+1;

}

for i:=low to high do

a[i]:=b[j];

}

}

# ANALYSIS:

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

In merge sort algorithm two recursive calls are made each call focus on n/2 elements of the list .after two recursive calls ,one call is made to combine two sublistsi.e, to merge all n elements.

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

a=2

b=2

f(n)=cn

### BY MASTERS THEOREM,

g(n)=pow(n,loga/logb)

g(n) =n

f(n)=cn let c=1 then f(n)=n

f(n)=g(n)

T(n)=O(nlogn)

QUICKSORT:

Quicksort technique based on divide and conquer strategy ,in this technique at every step each element is placed in the proper position.

It performs very well on longer lists,it works recursively by first selecting random pivot element from the array then it partitions the list into two elements that are less than the pivot .

The problem of sorting a given list is reduced to the problem of sorting two sublists and process continuous until the list is sorted.

RULES:

1. P = a[0]
2. i = a[1] , j = a[n-1]
3. After taking pivot element compare with pivot elements from left side ,if it is greater than the pivot element and it is considered as ‘i’ and from right side if it is smaller than the pivot element and it is considered as ‘j’.
4. If(i<j)then interchange array values a[i] & a[j]

Otherwise (i>=j) then interchange pivot and a[j] this procedure divides the list into two sublists a[0]............a[j-1] and a[j+1]..............a[n-1]

5.Apply same procedure for sublists which produces the list in sorted order.

0 1 2 3 4 5 6 7

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 65 | 75 | 80 | 85 | 60 | 55 | 50 | 45 |

0 1 2 3 4 5 6 7

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 65 | 45 | 80 | 85 | 60 | 55 | 50 | 75 |

0 1 2 3 4 5 6 7

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 65 | 45 | 50 | 85 | 60 | 55 | 80 | 75 |

0 1 2 3 4 5 6 7

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 65 | 45 | 50 | 55 | 60 | 85 | 80 | 75 |

0 1 2 3 4 5 6 7

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 60 | 45 | 50 | 55 | 65 | 85 | 80 | 75 |

|  |  |  |  |
| --- | --- | --- | --- |
| 60 | 45 | 50 | 55 |

|  |  |  |
| --- | --- | --- |
| 85 | 80 | 75 |

|  |
| --- |
| 65 |

|  |  |  |  |
| --- | --- | --- | --- |
| 55 | 45 | 50 | 60 |

|  |  |  |
| --- | --- | --- |
| 75 | 80 | 85 |

|  |
| --- |
| 65 |

|  |  |  |  |
| --- | --- | --- | --- |
| 50 | 45 | 55 | 60 |

|  |  |  |
| --- | --- | --- |
| 75 | 80 | 8`5 |

|  |
| --- |
| 65 |

|  |  |  |  |
| --- | --- | --- | --- |
| 45 | 50 | 55 | 60 |

|  |  |  |
| --- | --- | --- |
| 75 | 80 | 85 |

|  |
| --- |
| 65 |

QUICKSORT ALGORITHM:

Algorithm Quicksort(a,low,high)

{

If(low<=high)then

{

J:=partition(a,low,high);

Quicksort(a,low,high);

Quicksort(a,low,j-1);

Quicksort(a,j+1,high);

}

}

Algorithm partition(a,low,high)

{

Pivot:= a[low];

i:=low;

j:=high;

while(i<=j)do{

while(a[i]<=pivot)do{

i:=i+1;

while(a[j]>pivot)do

j:=j-1;

if(i<=j)then

swap(pivot,a[j]);

return j;

}

}

}

TIME COMPLEXITY:

T(N) = { a n = 1

T(n/2)+n n>1

}

T(N) = a T(n/b) + f(n)

f(n) = n;

a = 2;

b = 2;

g(n) = n^log^ab

= n^log^22

=n(1)

g(n) = n

f(n) = g(n) then O(g(n)logn) = O(nlogn)

**KARATSUBA METHOD:**

p = xz+yw

p = (x+y)(w+z)-xw-yz

p1=xw

p2=yz

p=(x+y)(w+z)-p1-p2

uv = ((x+y)(w+z)-p1-p2)10m+p2

Karatsuba method takes total 3 number of multiplications for solviing the above problem .

Hence the time complexity is T(n) =3T (n/2)+c.n

a=3 ; f(n)=c

b=2 ;

g(n) = nlogba

= nlog23

=> g(n) = n1.58

f(n) < g(n)

= O(n1.58)

EX: Multiply the following 4 digit numbers using karatsuba method

u = 2345 and v = 5678

sol:

u=2345

v=5678

u = 23\*102+45

v = 56\*102+78

=> m=n/2

=> uv= ((x+y)(w+z)-p1-p2)10m+p2

=> p1 = xw =(23)(56)=1288

=> p2 = yz = (45)(78)=3510

=> uv = 1288 \*104+((68)\*(134)-1288-3510)102 +3510

= 13314910

***STRASSEN’S MULTIPLICATION:***

Using Strassen’s Matrix multiplication algorithm, the time consumption can be improved a little bit.

Strassen’s Matrix multiplication can be performed only on **square matrices** where **n** is a **power of 2**. Order of both of the matrices are **n × n**.

Divide **X**, **Y** and **Z** into four (n/2)×(n/2) matrices as represented below −

**Z** =[IJKL] **X**=[ACBD] and **Y**=[EGFH]

Using Strassen’s Algorithm compute the following −

M1:=(A+C)×(E+F)

M2:=(B+D)×(G+H)

M3:=(A−D)×(E+H)

M4:=A×(F−H)

M5:=(C+D)×(E)

M6:=(A+B)×(H)

M7:=D×(G−E)

Then,

I:=M2+M3−M6−M7

J:=M4+M6

K:=M5+M7

L:=M1−M3−M4−M5

**Analysis:**

T(n)={c7xT(n2)+dxn2ifn=1otherwiseT(n)={cifn=17xT(n2)+dxn2otherwise

where *c* and *d* are constants

Using this recurrence relation,

we get

T(n)=O(nlog7)T(n)=O(nlog7)

Hence, the complexity of Strassen’s matrix multiplication algorithm is O(nlog7)O(nlog7).