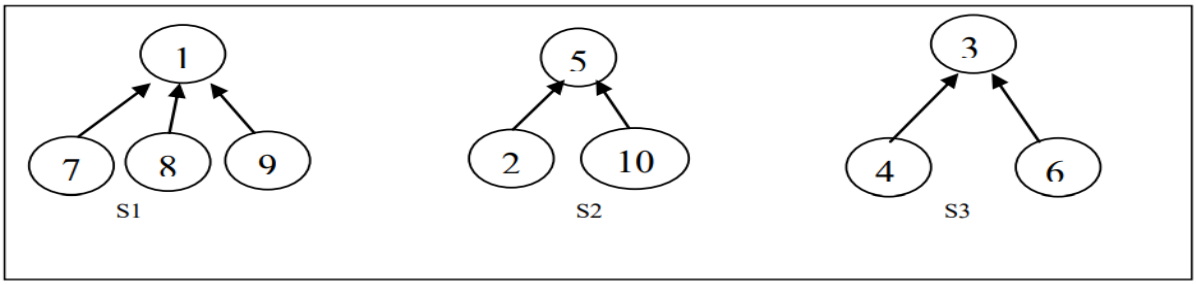
Module-2: Disjoint Sets, Divide and Conquer

2.1. Disjoint Sets

Disjoint Sets: If S_i and S_j, $i \neq j$ are two sets, then there is no element that is in both S_i and S_j.
For example: n=10 elements can be partitioned into three disjoint sets,

S_1= {1, 7, 8, 9}
S_2= {2, 5, 10}
S_3= {3, 4, 6}

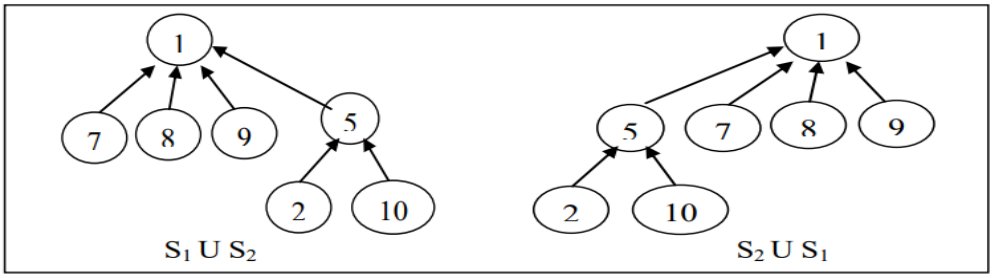
Tree representation of sets:



Disjoint set Operations:

- Disjoint set Union
- Find(i)

Disjoint set Union: Means Combination of two disjoint sets elements. Form above example $S_1 \cup S_2 = \{1,7,8,9,5,2,10\}$
For $S_1 \cup S_2$ tree representation, simply make one of the tree is a subtree of the other.



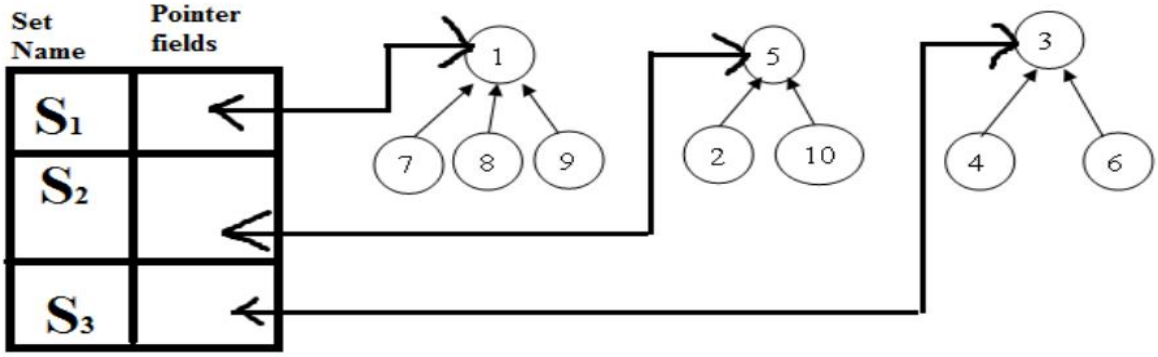
Find: Given element i, find the set containing i.
Form above example:

Find(4)→S_3

Find(1)→S_1
Find(10)→S_2

Data representation of sets:

Tress can be accomplished easily if, with each set name, we keep a pointer to the root of the tree representing that set.



For presenting the union and find algorithms, we ignore the set names and identify sets just by the roots of the trees representing them.
For example: if we determine that element 'i' is in a tree with root 'j' has a pointer to entry 'k' in the set name table, then the set name is just **name[k]**

For unite (**adding or combine**) to a particular set we use FindPointer function.
Example: If you wish to unite to S_i and S_j then we wish to unite the tree with roots
FindPointer (S_i) and FindPointer (S_j)
FindPointer→ is a function that takes a set name and determines the root of the tree that represents it.
For determining operations:
Find(i)→ 1^{St} determine the root of the tree and find its pointer to entry in setname table.
Union(i, j)→ Means union of two trees whose roots are i and j.

If set contains numbers 1 through n, we represents tree node
P[1:n].
n→Maximum number of elements.

Each node represent in array

i	1	2	3	4	5	6	7	8	9	10
P	-1	5	-1	3	-1	3	1	1	1	5

Find(i) by following the indices, starting at i until we reach a node with parent value -1.
Example: Find(6) start at 6 and then moves to 6's parent. Since P[3] is negative, we reached the root.

Algorithm for finding Union(i, j):	Algorithm for find(i)

Algorithm Simple union(i, j)
{
P[i]:=j; // Accomplishes the union
}
	Algorithm SimpleFind(i)
	{
	While(P[i]≥0) do i:=P[i];
	return i;
	}

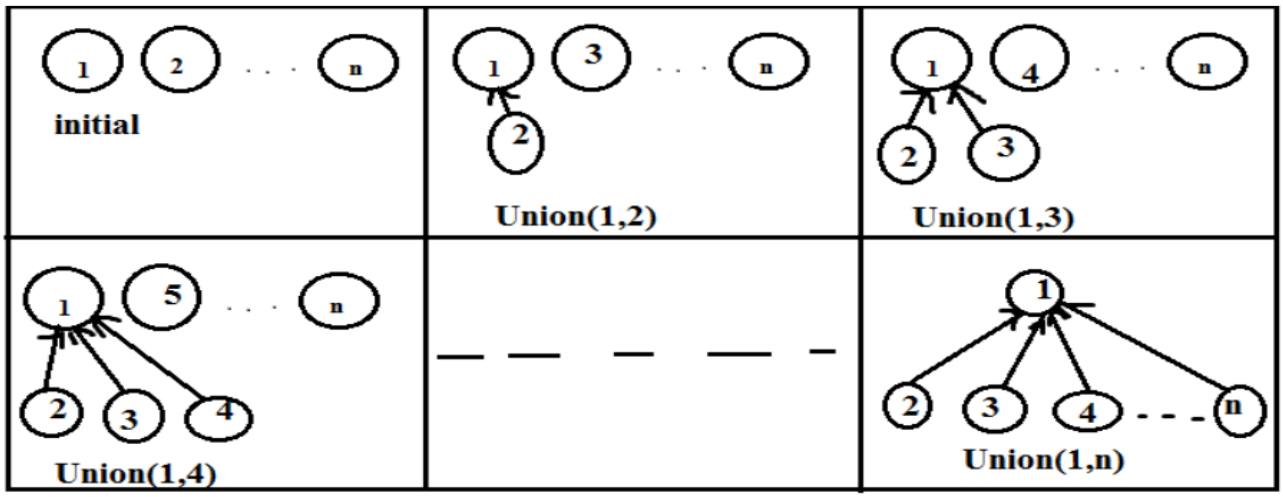
If n numbers of roots are there then the above algorithms are not useful for union and find.
For union of n trees→ Union(1,2), Union(2,3), Union(3,4),…..Union(n-1,n).
For Find i in n trees→ Find(1), Find(2),….Find(n).

Time taken for the union (simple union) is → O(1) (constant).
For the n-1 unions→ O(n).

Time taken for the find for an element at level i of a tree is → O(i).
For n finds → $O(n^2)$.

To improve the performance of our union and find algorithms by avoiding the creation of degenerate trees. For this we use a weighting rule for union(i, j)

Weighting rule for Union(i, j):
If the number of nodes in the tree with root 'i' is less than the tree with root 'j', then make 'j' the parent of 'i'; otherwise make 'i' the parent of 'j'.



Tree obtained using the weighting rule

Algorithm for weightedUnion(i, j)

```
Algorithm WeightedUnion(i,j)
//Union sets with roots i and j, i≠j
// The weighting rule, p[i]= -count[i] and p[j]= -count[j].
{
temp := p[i]+p[j];
if (p[i]>p[j]) then
{ // i has fewer nodes.
P[i]:=j;
P[j]:=temp;
}
else
{ // j has fewer or equal nodes.
P[j] := i;
P[i] := temp;
}
}
```

For implementing the weighting rule, we need to know how many nodes there are in every tree.
For this we maintain a count field in the root of every tree.
i→ root node
count[i]→ number of nodes in the tree.
Time required for this above algorithm is O(1) + time for remaining unchanged is determined by using **Lemma.**

Lemma:- Let T be a tree with **m** nodes created as a result of a sequence of unions each performed using WeightedUnion. The height of T is no greater than $\lfloor \log_2 m \rfloor + 1$.

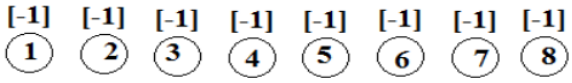
Collapsing rule: If 'j' is a node on the path from 'i' to its root and p[i]≠root[i], then set p[j] to root[i].

Algorithm for Collapsing find.

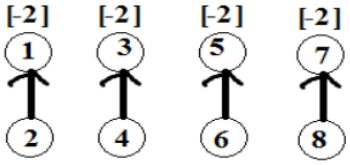
```
Algorithm CollapsingFind(i)
//Find the root of the tree containing element i.
//collapsing rule to collapse all nodes form i to the root.
{
r;=i;
while(p[r]>0) do r := p[r]; //Find the root.
While(i ≠ r) do // Collapse nodes from i to root r.
{
s:=p[i];
p[i]:=r;
i:=s;
}
return r;
}
```

Collapsing find algorithm is used to perform find operation on the tree created by WeightedUnion.

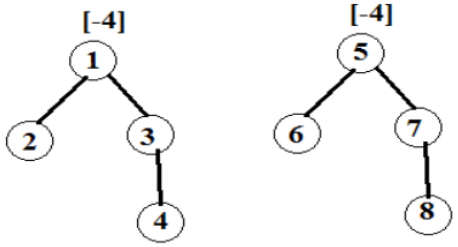
For example: Tree created by using WeightedUnion



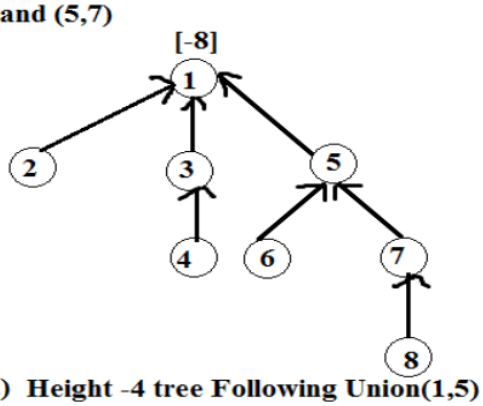
(a) initial height -1 tree



(b) Height -2 trees follwoing Union(1,2),(3,4),(5,6),(7,8)



(c) Height -3 trees following Union (1,3) and (5,7)



(d) Height -4 tree Following Union(1,5)

Now process the following eight finds: Find(8), Find(8),…….Find(8)
If SimpleFind is used, each Find(8) requires going up three parent link fields for a total of 24 moves to process all eight finds.

When CollapsingFind is uised the first Find(8) requires going up three links and then resetting two links. Total 13 movies requies for process all eight finds.

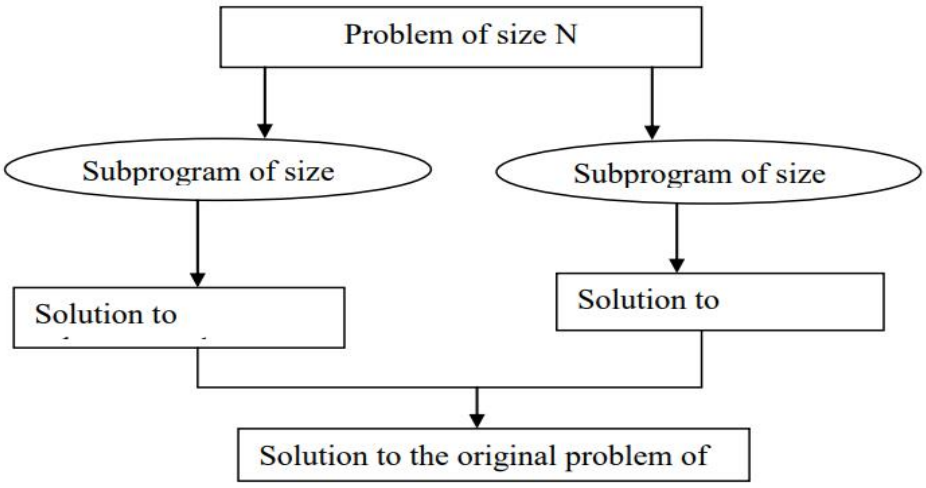
2.2. Divide and Conquer

General Method

In divide and conquer method, a given problem is,
i) Divided into smaller subproblems.
ii) These subproblems are solved independently.
iii) Combining all the solutions of subproblems into a solution of the whole.

If the subproblems are large enough then divide and conquer is reapplied.
The generated subproblems are usually of some type as the original problem.

Hence recurssive algorithms are used in divide and conquer strategy.



Pseudo code Representation of Divide and conquer rule for problem "P"

```
Algorithm DAndC(P)
{
if small(P) then return S(P)
else{
divide P into smaller instances P1,P2,P3…Pk;
apply DAndC to each of these subprograms; // means DAndC(P1), DAndC(P2)…..
DAndC(Pk)
return combine(DAndC(P1), DAndC(P2)….. DAndC(Pk));
}
}

//P➔Problem
//Here small(P)➔ Boolean value function. If it is true, then the function S is
//invoked
```

Time Complexity of DAndC algorithm:

$$T(n) = \begin{cases} T(1) & \text{if } n=1 \\ aT(n/b)+f(n) & \text{if } n>1 \end{cases}$$

a,b➔ contants.
This is called the **general divide and-conquer recurrence.**

Example for GENERAL METHOD:
As an example, let us consider the problem of computing the sum of n numbers a_0, ... a_{n-1}.
If n > 1, we can divide the problem into two instances of the same problem. They are sum of the first | n/2|numbers
Compute the sum of the 1st [n/2] numbers, and then compute the sum of another n/2 numbers. Combine the answers of two n/2 numbers sum.
i.e.,
$a_0 + \ldots + a_{n-1} = (a_0 + \ldots + a_{n/2}) + (a_{n/2} + \ldots + a_{n-1})$
Assuming that size n is a power of b, to simplify our analysis, we get the following recurrence for the running time T(n).
T(n)=aT(n/b)+f(n)

This is called the general **divide and-conquer recurrence**.
f(n)→ is a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions. (For the summation example, a = b = 2 and f (n) = 1.

Advantages of DAndC:
The time spent on executing the problem using DAndC is smaller than other method.
This technique is ideally suited for parallel computation.
This approach provides an efficient algorithm in computer science.

Master Theorem for Divide and Conquer
In all efficient divide and conquer algorithms we will divide the problem into subproblems, each of which is some part of the original problem, and then perform some additional work to compute the final answer. As an example, if we consider merge sort [for details, refer Sorting chapter], it operates on two problems, each of which is half the size of the original, and then uses O(n) additional work for merging. This gives the running time equation:

$T(n) = 2T(\frac{n}{2}) + O(n)$

The following theorem can be used to determine the running time of divide and conquer algorithms. For a given program or algorithm, first we try to find the recurrence relation for the problem. If the recurrence is of below form then we directly give the answer without fully solving it.

If the reccurrence is of the form $T(n) = aT(\frac{n}{b}) + \Theta(nk\log^p n)$, where a >= 1, b > 1, k >= O and p is a real number, then we can directly give the answer as:

1) If $a > b^k$, then $T(n) = \Theta(n^{\log_b^a})$

2) If $a = b^k$
a. If p > -1, then $T(n) = \Theta(n^{\log_b^a} \log^{p+1} n)$
b. If p = -1, then $T(n) = \Theta(n^{\log_b^a} \log \log n)$
c. If p < -1, then $T(n) = \Theta(n^{\log_b^a} s)$

3) If $a < b^k$
a. If p >= 0, then $T(n) = \Theta(n^k \log^p n)$
b. If p < 0, then $T(n) = 0(n^k)$

Applications of Divide and conquer rule or algorithm:

- Binary search,
- Quick sort,
- Merge sort,
- Strassen's matrix multiplication.

Binary search or Half-interval search algorithm:

1. This algorithm finds the position of a specified input value (the search "key") within an array sorted by key value.
2. In each step, the algorithm compares the search key value with the key value of the middle element of the array.
3. If the keys match, then a matching element has been found and its index, or position, is returned.
4. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the **left** of the middle element or, if the search key is greater, then the algorithm repeats on sub array to the **right** of the middle element.
5. If the search element is less than the minimum position element or greater than the maximum position element then this algorithm returns not found.

Binary search algorithm by using recursive methodology:

Program for binary search (recursive)	Algorithm for binary search (recursive)
int binary_search(int A[], int key, int imin, int imax)	**Algorithm binary_search(A, key, imin, imax)**

```
{
if (imax < imin)
   return array is empty;
if(key<imin || K>imax)
   return element not in array list
 else
   {
    int imid = (imin +imax)/2;
    if (A[imid] > key)
    return binary  search(A, key, imin, imid-1);
    else if (A[imid] < key)
    return binary_search(A, key, imid+1, imax);
    else
    return imid;
   }
}
```

```
{
 if (imax < imin) then
return "array is empty";
 if(key<imin || K>imax) then
return "element not in array list"
 else
   {
    imid = (imin +imax)/2;
    if (A[imid] > key) then
return binary  search(A, key, imin, imid-1);
    else if (A[imid] < key) then
return binary_search(A, key, imid+1, imax);
    else
    return imid;
   }
}
```

Time Complexity:
Data structure:- Array

For successful search	Unsuccessful search
Worst case→ O(log n) or θ(log n) Average case→ O(log n) or θ(log n) Best case→ O(1) or θ(1)	θ(log n):- for all cases.

Binary search algorithm by using iterative methodology:

Binary search program by using iterative methodology:	Binary search algorithm by using iterative methodology:

int binary_search(int A[], int key, int imin, int imax)
{
while (imax >= imin)
{
int imid = midpoint(imin, imax);
if(A[imid] == key)
return imid;
else if (A[imid] < key)
imin = imid + 1;
else
imax = imid - 1;
}
}
	Algorithm binary_search(A, key, imin, imax)
	{
	While < (imax >= imin)> do
	{
	int imid = midpoint(imin, imax);
	if(A[imid] == key)
	return imid;
	else if (A[imid] < key)
	imin = imid + 1;
	else
	imax = imid - 1;
	}
	}

Merge Sort

Merge sort is a classic divide-and-conquer algorithm used for sorting an array. It works by recursively dividing the array into halves, sorting each half, and then merging the sorted halves back together. Merge sort does not typically involve dynamic programming, but rather uses a recursive approach. However, I can provide a detailed explanation of the merge sort algorithm and its implementation.

Merge Sort Algorithm

1. Divide:

   Divide the unsorted array into approximately two equal parts.

2. Conquer:

   Sort both parts recursively.

3. Combine:

   Merge both sorted parts to generate the complete sorted array.

Implementation Steps

1. Base Case:

   The array was already sorted when the elements are 0 or 1.

2. Recursive Case:

Split the array into two halves.

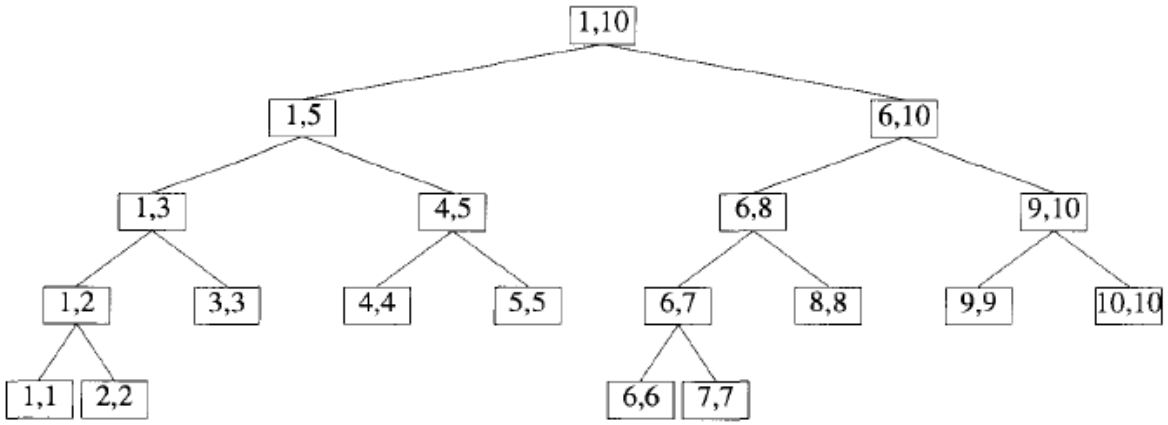
Recursively sort each half.

Merge the two sorted halves.

Algorithm:

```
1   Algorithm MergeSort(low, high)
2   // a[low : high] is a global array to be sorted.
3   // Small(P) is true if there is only one element
4   // to sort. In this case the list is already sorted.
5   {
6       if (low < high) then  // If there are more than one element
7       {
8           // Divide P into subproblems.
9               // Find where to split the set.
10                  mid := ⌊(low + high)/2⌋;
11          // Solve the subproblems.
12              MergeSort(low, mid);
13              MergeSort(mid + 1, high);
14          // Combine the solutions.
15              Merge(low, mid, high);
16      }
17  }
```

Example:



Matrix Multiplication using Strassen's Method

Strassen suggested a DAC strategy-based matrix multiplication that requires fewer multiplications than the conventional one. Using Strassen's method, the matrix multiplication can be defined as follows:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

$$C11 = S1 + S4 - S5 + S7$$
$$C12 = S3 + S5$$
$$C21 = S2 + S4$$
$$C22 = S1 + S3 - S2 + S6$$

Where,

$$S1 = (A11 + A22) * (B11 + B22)$$
$$S2 = (A21 + A22) * B11$$
$$S3 = A11 * (B12 - B22)$$
$$S4 = A22 * (B21 - B11)$$
$$S5 = (A11 + A12) * B22$$
$$S6 = (A21 - A11) * (B11 + B12)$$
$$S7 = (A12 - A22) * (B21 + B22)$$

Let us check if it is the same as the conventional approach.

$$\begin{aligned} C12 &= S3 + S5 \\ &= A11 * (B12 - B22) + (A11 + A12) * B22 \\ &= A11 * B12 - A11 * B22 + A11 * B22 + A12 * B22 \\ &= A11 * B12 + A12 * B22 \end{aligned}$$

This is the same as C_{12} derived using the traditional approach. Similarly, we can derive all C_{ij} for Strassen's matrix multiplication.

Complexity Analysis of Matrix Multiplication using DAC approach.

The conventional approach employs eight multiplications for two 2 x 2 size matrices. Strassen's approach employs seven multiplications for a size 1 x 1 matrix, which in turn finds the multiplication of 2 x 2 matrices using addition. In short form, to solve the problem of size n, Strassen's approach creates seven problems of size (n – 2). The recurrence equation for Strassen's approach is given as,

$$T(n) = 7.T(n/2)$$

One multiplication only needed for two matrices of size 1 x 1; thus, T(1) = 1 in the base case. We will find the solution by replacing $n = (\frac{n}{2})$ iteratively in the previous equation.

$$T(n/2) = 7.T(n/4)$$

$$\Rightarrow T(n) = 72.T(n/22)$$
$$T(n) = 7k.T(2k)$$
$$Let's\ assume\ n = 2k \Rightarrow k = log2\ n$$
$$T(n) = 7k\ .T(2k/2k)$$
$$= 7k\ .T\ (1)$$
$$= 7k$$
$$= 7log2\ n$$
$$= nlog2\ 7$$
$$= n2.81 < n3$$

Thus, Strassen's matrix multiplication algorithm's running time is $O(n^{2.81})$, less than the traditional approach's cubic order. When n is larger, the running time difference becomes significant.