DIVIDE-AND-CONQUER STRATEGY

Sometimes, when a problem seems too large or complicated that it looks impossible to solve or at least very difficult to do so, it helps to break down the problem into subproblems, solve the individual small subproblems, and combine their solutions to get the solution to the original problem. The formal definition is given below:

Given a function to compute on n inputs, the divide-and-conquer strategy involves splitting the inputs into k distinct subsets, 1 < k <= n yielding k subproblems. These subproblems are to be solved and then a method must be found to combine these subsolutions into a solution of the whole problem. If the subproblems are found to be still large, the divide-and-conquer strategy may again be applied to the subproblems.

Often, the subproblems are of the same type as the original problem, in which case the divide-and-conquer method used is inherently expressed by a recursive procedure.

The simplest case of divide-and-conquer is to divide the set into two sets (i.e., k =2). The following control abstraction will give a clearer understanding of the divide-and-conquer strategy.

```
procedure DANDC( p, q)
        global n, A(1: n); integer  m, p,q;
                                  // 1<= p <= q <=n  //
        if SMALL(p,q)
                then return (G(p,q))
                else  m → DIVIDE(p,q)
                                  // p <=m <q  //
                 return(COMBINE( DANDC(p,m), DANDC(m+1, q)))
        endif
end DANDC
```

Fig. 1. Control abstraction for divide-and-conquer

SMALL(p,q) is a Boolean valued function that determines if the input size q - p +1 is small enough so that the answer can be computed without splitting; if so G is invoked.

DIVIDE(p, q) returns an integer that specifies where the input is to be split.

COMBINE(x, y) is a function that determines the solution to A(p:q) using the solutions x and y to the two subproblems A(p:m) and A(m+1, q).

If the sizes of the two subproblems are approximately equal, then the computing time, T(n) of DANDC is naturally described by the recurrence relation

T(n) = g(n), if n is small
T(n) = 2T(n/2) + f(n), otherwise

where T(n) is the time for DANDC on n inputs,
g(n) is the time to solve for small inputs,
f(n) is the time for DIVIDE and COMBINE.

Problem Specification

Several problems can be solved with the help of the divide-and-conquer strategy represented by a recursive definition. To come up with a recursive definition for a problem, first, we need to specify the problem.

The following examples show how a problem is specified:
Example 1:
problem statement: Find the minimum of n elements
input: L={e_1, e_2, …., e_n}
output: e; e <= e_i for all i=1,..,n

Example 2:

problem statement: Find the maximum of n elements
input: L={e_1, e_2, …., e_n}
output: e; e >= e_i for all i=1,..,n

Example 3:
problem statement: Search for an element, x in a list of n elements
input: x, L={e_1, e_2, …., e_n}
output: i (1<=i<=n) if x is found at i
0 otherwise

Example 4:
problem statement: Sort a list of n elements in non-descending order
input: L_1={e_1, e_2, …., e_n}
output: L_2={o_1, o_2, …., o_n}, ; o_i <= o_{i+1} for i<=n-1

Development of a Recursive Definition
Example 1:
problem statement: Find the minimum of n elements
input: L={e_1, e_2, …., e_n}
output: e; e <= e_i for all i=1,..,n; e ϵ L

```
min(L)
{
if |L|==1, then return e1 ;// e1 is the first element in L
split L into L1 and L2 ;
        //(L1 and L2 may be of same or different sizes, such //that (L1 || L2 = L)
o1 = min (L1)
o2 = min (L2)
if o1 <= o2 , return o1  //combining results
else return o2
}
```

Use the recursive definition to determine a recurrence relation and solve the relation.
NOTE: Only the no. of element comparisons is considered. The split is in the middle.

$T(n) = 0$ if $n=1$
$T(n) = T(n/2)+T(n/2)+1$ if $n>1$

where T(n) is the time taken (represented by the no. of element comparisons) by the algorithm.

Example 4:
problem statement: Sort a list of n elements in non-descending order
input: $L_1=\{e_1, e_2, \ldots, e_n\}$
output: $L_2=\{o_1, o_2, \ldots, o_n\}$, ; $o_i <= o_{i+1}$ for $i<=n-1$
Sort(L)
{
if |L|==1, return L;
split L into L_1 and L_2 using k ;
//($L_1=\{e_1, e_2, \ldots, e_k\}$ $L_2=\{e_{k+1}, e_{k+2}, \ldots, e_n\}$
Sort(L_1)
Sort(L_2)
Merge(L_1,L_2) //merge the sorted subsets
}

$T(n) = 0$ if $n=1$
$T(n) = T(n/k)+T(n-k)+n$ if $n>1$
n is for merging

Example 3:
problem statement: Search for an element, x in a list of n elements
input: x, $L=\{e_1, e_2, \ldots, e_n\}$
output: i ($1<=i<=n$) if x is found at i
-1 otherwise

Search(x,L,low,high) //low and high are the 1st and last indices of list L
{
if |L|==1 // of if L[low]==L[high], contains only 1 element
        if x== L[low], return low;
        else return -1;
//split L into L_1 and $L_{2;}$ in the middle
mid = (low+high)/2;
i = Search(x,L_1, low,mid);
j = Search(x,L_2, mid+1, high);
if (i !=-1) return i;
else return j;
}

problem statement: Search for an element, x in a sorted list of n elements
input: x, L={e_1, e_2, …., e_n}, e_i <= e_{i+1}, for i<=n-1
output: i (1<=i<=n) if x is found at i
                0 otherwise

BinSearch(x,L)
{
if |L|==0 return 0;
split L into 3 parts: L_1 , middle element and $L_{2;}$
if x== middle element   return mid (i.e., the position of middle element);
else
        if x < middle element
                BinSearch(x,L_1);
        else
                BinSearch(x,L_2);
}

T(n) = 1   if n=1
T(n) = T(n/2)+1   if n>1

1 is for splitting and element comparison

Three methods for solving a recurrence relation:

i) Substitution method: Needs guesswork

ii) Recursion Tree method

iii) Master Theorem method

Master Theorem

Given a recurrence of the form

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

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be an asymptotically positive function, and let $T(n)$ be a function over the positive numbers defined by the recurrence

Then *T(n)* can be bounded asymptotically as follows:

Case a) If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 1$, then $T(n) = \Theta(n^{\log_b a})$.

Case b) if $f(n) = \Theta(n^{\log_b a} log^k n)$ with* $k>=0$, then $T(n) = \Theta(n^{\log_b a} log^{k+1} n)$.

Case c) If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 1$, (and $af(n/b) <= cf(n)$ for some $c < 1$ for all sufficiently large n), $T(n) = \Theta(f(n))$

NOTE: *most of the time, k=0.

Master Theorem (simplified)[when k=0 in the above discusion]

Given a recurrence of the form

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

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be a function over the positive numbers defined by the recurrence

Then *T(n)* can be bounded asymptotically as follows:

Case a) If $f(n) = O(n^{\log_b a-\epsilon})$ for some $\epsilon > 1$, then $T(n) = \Theta(n^{\log_b a})$

Case b) if $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log_2 n)$

Case c) If $f(n) = \Omega(n^{\log_b a+\epsilon})$) for some $\epsilon > 1$, (and $af(n/b) <= cf(n)$ for some c <1 for all sufficiently large n), $T(n) = \Theta(f(n))$

Case a) checks whether:

i) $f(n)$ is smaller than $n^{\log_b a}$

ii) $f(n)$ is polynomially smaller than $n^{\log_b a}$ by a factor of n^{ϵ}

Case b) checks whether:

i) $f(n)$ is same as $n^{\log_b a}$

Case c) checks whether:

i) $f(n)$ is larger than $n^{\log_b a}$

ii) $f(n)$ is polynomially larger than $n^{\log_b a}$ by a factor of n^{ϵ}

iii) regularity constraint is satisfied. It is satisfied by most of the polynomially bounded functions

So, compare $f(n)$ with $n^{\log_b a}$**.**

Case a) If $f(n)$ is smaller than $n^{\log_b a}$**, then** $T(n) = \Theta(n^{\log_b a})$

Case b) If $f(n)$ is equal to $n^{\log_b a}$ **, then** $T(n) = \Theta(n^{\log_b a} \log_2 n)$

Case c) If $f(n)$ is larger than $n^{\log_b a}$**, then** $T(n) = \Theta(f(n))$

Binary Search:

T(1) = 1,

T(n) = T(n/2) + 1, when n > 1

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

Here,

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

$n^{\log_b a} = n^{\log_2 1} = n^0 = 1$

Here, f(n) is equal to $n^{\log_b a}$. Hence T(n) = $\Theta(n^{\log_b a} \log_2 n) = \Theta(\log_2 n)$

Ex2.

T(n) = 3T(n/4) + cn^2

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

a= 3 >=1, b=4>1 , f(n) = cn^2 = O(n^2)

$n^{\log_b a} = n^{\log_4 3} = n^{0.8}$

$T(n) = \Theta(f(n)) = \Theta(n^2)$

T(n) = O(n^2)

Ex. 3.

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

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

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

$n^{\log_b a} = n^{\log_2 2} = n^1 = n$

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

- Ningrinla

Substitution Method:

Ex. 4.

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

Guess the solution first:
T(n) = O(n), i.e., T(n) <= dn (1)

Suppose, it is true for n/2.
Then, by substituting using eqn. (1),
we get,

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

T(n) <= d(n/2) + cn since T(n/2) <= d(n/2) from eqn. (1)
= n(d/2 + c)
<= dn if (d/2 + c) <= d

(d/2 + c) <= d
i.e., d – d/2 >= c
i.e., d/2 >= c
i.e., d >= 2c

Choose d such that d >=2c. Then T(n) = O(n) since there exists a constant
d such that T(n) <=dn for n >=1.

Binary Search

A good example of the divide-and-conquer strategy is the binary search algorithm. Here, the problem is to search for an element among a set of n elements sorted in the nondecreasing order.

```
BinSearch(x,L)
{
if |L|==0 return 0; //returns 0 if the element is not found.
```

split L into 3 parts: L_1 , middle element and $L_{2;}$
if x== middle element return mid (i.e., the position of middle element);
else
        if x < middle element
                BinSearch(x,L_1);
        else
                BinSearch(x,L_2);
}

//Recursive binary search – more detailed
BinSearch1(x,low,high,L)
{
if low > high return 0; // returns 0 if element is not found.
mid = (low+high)/2; //integer division
if x < L(mid)
        BinSearch1(x, low, mid-1, L);
else
        if x > L(mid)
                BinSearch1(x, mid+1, high, L);
        else //i.e., x == L(mid)
                return mid; // x found at position, mid
}

Initial call to BinSearch1(x, low, high, L) is BinSearch1(x, 1, n, L)
Where x is the search element, L is the sorted set of n elements.

//Iterative Binary search
BinSearch2(x,low,high,L)
{ //returns the position where x is found or 0 if x is not found
low = 1; high =n;

```
while  (low <= high) do
        mid = (low+high)/2; //integer division
        if x < L(mid)
                high = mid -1;
        else
                if x > L(mid)
                        low = mid+1;
                else   //i.e., x == L(mid)
                        return mid; // x found at position, mid
end while
return 0
}
```

Theorem:

The time for a successful search is O(log n) and for an unsuccessful search is θ(log n).

(NOTE: Watch the video for the explanation (for COVID days). Done in the class)

Mergesort

Another example of the divide-and-conquer strategy is the mergesort algorithm, which sorts a set of n elements. The idea is to split the set of elements into two equal-sized sets, sort these two sets individually, and merge the results to get a sorted sequence of the whole. Procedure MERGESORT does the splitting and calls another procedure, MERGE which merges two sorted sets.

```
// merge function takes two intervals
// one from low to high
// second from mid+1, to high
// and merge them in sorted order

merge(low, mid, high)
begin
        //Create a temporary array, B
        // index variables for both intervals and for temp
        i = low, j = mid+1, k = low;

        // traverse both subarrays and add smaller of both elements in B
        while(i <= mid && j <= high) do
                if(A[i] <= A[j]) then
                        B[k] = A[i];
                        i = i + 1;
                else
                        B[k] = A[j];
                        j = j + 1;
                endif
                k = k + 1;
        end while

        //Add remaining elements to B
        If (k > mid) then  //all the elements in the 1st subarray are already in B
                // so, add the remaining elements in the 2nd subarray to B
                while (j <= high) do
                        B[k] = A[j];
                        k = k + 1; j = j + 1;
```

```
                end while
        else //all the elements in the 2nd subarray are already in B
                // so, add the  remaining elements in the 1st subarray to B
                while (i <= mid) do
                        B[k] = A[i];
                        k = k + 1; i = i + 1;

                end while
        endif
        // copy back from B to A
        for k = low to high do
                A[k] = B[k];
        end for
end

// A is an array of integer type
// start and end are the starting and ending index of the current interval of Arr

mergeSort(low, high)
//A[1:n] is an array of n elements. Initial call will be mergeSort(1,n)
begin
        if (low < high) then //if there are at least 2 elements, then prob is big
                mid = (low + high) / 2;
                mergeSort (low, mid);
                mergeSort(mid+1, high);
                merge(low, mid, high);
        endif
end
```

T(n) = 2T(n/2) + cn
The time complexity is O(nlog n).

Disadvantages of the procedure, MERGESORT:

- It uses 2n locations for the elements. This is because an additional array B is used. This use can be avoided by using an array, LINK of integers. Instead of moving elements from A to B, the links (contents of array, LINK) only are updated.

- Stack space is required because of recursion. The algorithm works top-down and splits in the middle. So the maximum depth of recursion is proportional to log n. The need for stack space can be avoided if we build an algorithm that works 'bottom-up'.

- For small-size sets, most of the time will be spent processing the recursion instead of sorting. To improve on this, we can use another sorting procedure (which works very fast for small-size sets) for small size sets.

Quicksort

The quicksort algorithm is another example of the divide-and-conquer strategy. Both mergesort and quicksort divide the set of elements to be sorted into two subsets but they differ in the way they divide the set of elements. In mergesort, the set of elements A(1:n) is divided at its midpoint into subsets which are independently sorted and later merged. In quicksort, the division into two subsets is made in such a way that the sorted subsets need not be later merged.

In quicksort, we first start by choosing any one of the elements (usually the first element in the set to be sorted) as the **partitioning** element. Then, we partition the set into two subsets; the left subset and the right subset with the partitioning element in the middle. The partition is done such that the elements in the left subset are less than or equal to the partitioning element, and the elements in the right subset are greater than or equal to the partitioning element. At this point, we can say that the partitioning element has been placed in its proper position in the final sorted order(i.e., the sorted order we want finally).

We again take the left subset, choose a partitioning element and so on. Similarly with the right subset. We can apply the procedure recursively until all the elements are found to be placed in their proper positions in the final sorted order; in other words, until the whole set if finally sorted.

```
/**
//Quicksort algorithm by C. A. R. Hoare (there is another by Lomuto)
* The main function that implements quick sort.
An array A[1:n] of elements to be sorted. The initial call to quicksort will
be quicksort(1,n).
*/
quickSort(low, high)
begin
    if (low < high) then //set is still big
        // pivot_position is the correct position of the pivot element.
        // A is partitioned such that the pivot element is placed in its
        //correct position in A.
        pivot_position = partition(low, high);

        quickSort(low, pivot_position - 1);  // sort subarray left of pivot
        quickSort(pivot_position + 1, high); // sort subarray right of pivot
    endif
end
```

```
/**
* The function selects the first element as pivot element, places that pivot
element correctly in the array in such a way
* that all the elements to the left of the pivot are lesser than or equal to
the pivot and
* all the elements to the right of pivot are greater than or equal to it.
* returns position of pivot element after placing it correctly in sorted array
*/
partition(low, high)
//A[n+1] = +α;
begin
    j = high + 1;
    i = low; True = 1;
    // pivot - Element at the left most position
    pivot = A [low]; //choose the first element as the pivot.
    while (True) do //forever loop
        do
                i = i + 1;   //loop (move i right) until A[i]>=pivot
        while (A[i] < pivot);
        do
                j = j - 1; //loop (move j left) until A[i]<=pivot
        while (A[j] > pivot);
```

```
        if (i < j) then
                interchange (A[j], A[i])
        else exit; //correct position of pivot element found
    endwhile
    // i.e., i >= j
    //correct position of pivot found, i.e., j
    A[low] = A[j];
    A[j] = pivot; return j;
end
```

Try out on this set:

5 2 6 1 3 5

5 2 6 1 3 5
3 2 6 1 5 5

Time complexity analysis of Quicksort algorithm:
Assumptions:
1) Count only the number of element comparisons C(n).
The frequency count of other operations is the same order as C(n)
2) The n elements to be sorted are distinct
3) The pivot (partitioning) element is randomly chosen.
RANDOM(i,j) generates a random integer in the interval [i,j]. Replace the statements pivot = A[low]; i = low by the following statements:
i = RANDOM(low,high); pivot = A[i]; A[i] = A[low]; i = low;

Case 1: Worst case analysis
The worst case data for Quicksort: It is when the elements are already in sorted order (non-decreasing or non-increasing) and the smallest element is chosen as the pivot.

The number of element comparisons in each case of partition is at most high-low+2. (i.e., in the first call to partition(1,n), n+1 comparisons).
(If the elements are not distinct, then at most high-low+3 comparisons may be made.

Let r be the total number of elements in all the calls to partition() at any level of recursion. n =6

1 2 3 4 5 6 partition(1,6); r= 6
2 3 4 5 6 partition(2,6); r= 5
3 4 5 6 partition(3,6); r= 4
4 5 6 partition(3,6); r= 3
5 6 partition(5,6); r= 2
6

At level 1, only 1 call to partition() is made. So, r = n.
At level 2, at most 2 calls to partition() is made. So, r = n-1.
At each level, r is at least one less than the r at the previous level as the partitioning element(s) get eliminated.

At each level, O(r) element comparisons are made by partition().
In the worst case, only one element will be eliminated at each level.
Thus, the worst case time complexity ($C_W(n)$) is the sum of r as r varies from 2 to n.
$C_W(n) = 2 + 3 + 4 + \ldots\ldots + n = (n(n+1)/2) - 1 = n^2/2 + n/2 - 1 = O(n^2)$

Case 2: Best case analysis
x x x x x x x x x x x
x x x x x x x x x x

The best case data is when the recursion tree generated is of the smallest height. That happens when a set (or subset) is always partitioned in the middle.
Hence, solving the following recurrence gives the Best case time complexity.
$T(n) = 2T(n/2) + cn$
The best case time complexity is O(nlog n).

Case 3: Average case analysis (Fundamentals of Algos. By Horowitz and Sahni, pp. 171-172).

Let $C_A(n)$ be the running time in the average case.

The pivot element (v) in the call to partition has an equal probability of being the ith smallest element in the set being sorted. Hence, the number of element comparisons associated with the two subsets to be sorted after the first call to partition (partition(1,n)) could be either one of the following:

1) $C_A(0)$ and $C_A(n-1)$, when the pivot element is the 1^{st} smallest element
2) $C_A(1)$ and $C_A(n-2)$, when the pivot element is the 2^{nd} smallest element
3) $C_A(2)$ and $C_A(n-3)$, when the pivot element is the 3^{rd} smallest element

.....

.....

n) $C_A(n-1)$ and $C_A(0)$, when the pivot element is the largest element

Therefore, $C_A(n)$ is given by the following recurrence relation:

$$C_A(n) = n + 1 + \frac{1}{n} \sum_{1 \le k \le n} [C_A(k-1)) + C_A(n-k)] \qquad (3.6)$$

The number of element comparisons required by Partition on its first call is $n+1$. Note that $C_A(0) = C_A(1) = 0$. Multiplying both sides of (3.6) by n, we obtain

$$nC_A(n) = n(n+1) + 2[C_A(0) + C_A(1) + \cdots + C_A(n-1)] \qquad (3.7)$$

Replacing n by $n-1$ in (3.7) gives

$$(n-1)C_A(n-1) = n(n-1) + 2[C_A(0) + \cdots + C_A(n-2)]$$

Subtracting this from (3.7), we get

$$\begin{aligned} nC_A(n) - (n-1)C_A(n-1) &= 2n + 2C_A(n-1) \\ \text{or} & \\ C_A(n)/(n+1) &= C_A(n-1)/n + 2/(n+1) \end{aligned}$$

Repeatedly using this equation to substitute for $C_A(n-1), C_A(n-2), \ldots$, we get

$$\begin{aligned} \frac{C_A(n)}{n+1} &= \frac{C_A(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &= \frac{C_A(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &\vdots \\ &= \frac{C_A(1)}{2} + 2\sum_{3 \le k \le n+1} \frac{1}{k} \\ &= 2\sum_{3 \le k \le n+1} \frac{1}{k} \end{aligned} \qquad (3.8)$$

Since

$$\sum_{3 \le k \le n+1} \frac{1}{k} \le \int_2^{n+1} \frac{1}{x}\, dx = \log_e(n+1) - \log_e 2$$

(3.8) yields

$$C_A(n) \le 2(n+1)[\log_e(n+2) - \log_e 2] = O(n \log n)$$

Strassen's matrix multiplication

A and B are two nxn matrices. C=AB is also an nxn matrix whose element at position (i,j) is formed by multiplying the ith row of A and the jth column of B.

$$C(i,j) = \sum_{k=1}^{n} A(i,k)B(k,j)$$

```
//C has been initialized to 0..
for i = 1 to n do
        for j = 1 to n do
                for k = 1 to n do
                        C(i,j) =  C(i,j) +A(i,k)*B(k,j)
                done
        done
done
```

Time taken with the conventional method is $\Theta(n^3)$.

Can I use the Divide-and-Conquer strategy to solve the matrix multiplication problem?

Assume for simplicity that n is a power of 2 (i.e., $n= 2^k$ for some value of k). If n is not a power of 2. We can use some transformation so that it becomes a power of 2. Add 0's to the matrices, A and B such that the order becomes a power of 2.

A

1	4	3
6	5	7

2x3 matrix converted to a 4x4 matrix using 0's.

1	4	3	0
6	5	7	0
0	0	0	0

0	0	0	0

C= AB, C's order is (2X2); C(1,1) = 1X1+4X2+ 3x3 = 18; C(1,2)=1x2+4X1+3X2=12; C(2,1)=6X1+5X2+7X3= 37; C(2,2)= 31

B

1	2
2	1
3	2

3x2 matrix converted to a 4x4 matrix using 0's.

1	2	0	0
2	1	0	0
3	2	0	0
0	0	0	0

Now, n = 4 (a power of 2)

Divide A and B into four square matrices. Each matrix has dimension, n/2 x n/2.

In this example,

A_{11} = | 1 | 4 | | A_{12} = | 3 | 0 |
|---|---|---|---|---|---|
| 6 | 5 | | | 7 | 0 |

A_{21} = | 0 | 0 | | A_{22} = | 0 | 0 |
|---|---|---|---|---|---|
| 0 | 0 | | | 0 | 0 |

In this example,

B_{11} = | 1 | 2 | | B_{12} = | 0 | 0 |
|---|---|---|---|---|---|
| 2 | 1 | | | 0 | 0 |

B_{21} = | 3 | 2 | | B_{22} = | 0 | 0 |
|---|---|---|---|---|---|
| 0 | 0 | | | 0 | 0 |

Then, C = AB can be computed as below:

A_{11}	A_{12}	X	B_{11}	B_{12}	=	C_{11}	C_{12}
A_{21}	A_{22}		B_{21}	B_{22}		C_{21}	C_{22}

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

$C_{12} = A_{11}\ B_{12} + A_{12}\ B_{22}$
$C_{21} = A_{21}\ B_{11} + A_{22}\ B_{21}$
$C_{22} = A_{21}\ B_{12} + A_{22}\ B_{22}$
Here, $A_{11}\ B_{11}$, and so on are calculated using the conventional method (since the problem is small enough).

$A_{11}\ B_{11}$ =

1+8	2+4
6+10	12+5

If n<=2, (i.e., the problem is small enough), we use the conventional method to compute C.
If n>2, (i.e., the problem is big), we use the divide-and-conquer strategy, as above.

Time taken is given by the following recurrence:
T(n) = b if n<=2
T(n) = 8T(n/2) + cn^2 if n>2

There are 4 additions. Each addition is element-wise addition of two square (n/2)x(n/2) matrices.

for i = 1 to n/2 do	$\Omega(n)$	O(n)
for j = 1 to n/2 do	$\Omega(n^2)$	$O(n^2)$
P(i,j)=S(i,j) + T(i,j)	$\Omega(n^2)$	$O(n^2)$
done		
done		

Time taken for each addition is $\Omega(n^2)$ and $O(n^2)$. 4 X $O(n^2)$ = $O(n^2)$ =cn^2

Here, cn^2 is contributed by the additions.
T(n) = e if n<=2
T(n) = 8T(n/2) + cn^2 if n>2

Here, b=2 and a =8. So, master theorem can be applied.
f(n) = cn^2

$(n^{\log_b a}) = n^{\log_2 8} = n^3$

$T(n) = \Theta(n^{\log_b a})$
Thus $T(n) = \Theta(n^3)$

Strassen improved on this divide-and-conquer method to reduce the number of multiplications to 7 and 18 additions or subtractions.

$P = (A_{11} + A_{22})\ (B_{11} + B_{22})$
$Q = (A_{21} + A_{22})\ B_{11}$

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

Time taken by Stassen's algorithm:

$T(n) = b$ if $n<=2$
$T(n) = 7T(n/2) + cn^2$ if $n>2$

$f(n) = cn^2$
$(n^{\log_b a}) = n^{\log_2 7} = n^{2.81}$

$T(n) = \Theta(n^{\log_b a})$ where b=2 and a =7.
Thus $T(n) = \Theta(n^{2.81})$

```
(max,min) MaxMin (A[1..n]) //returns the max. and the min. from a list of n elements.
{
 If (n==1) //…if problem size is small.
        return (A[1], A[1]);
 else {
        (max,min) = MaxMin(A[2..n]); // get the max. and min. from A[2..n]
        If (A[1]> max)
                max = A[1];
        else if (A[1] < min)
                min = A[1];
        return (max, min);
 }
}
```

Recurrence relation that represents the running time:
Count only the element comparisons.
T(n) = The running time of MaxMin when A consists of n elements.

T(n) = 0 if n=1

T(n) = T(n-1) + c if n > 1 T(n) = a T(n/b) + f(n)
