Unit II - DIVIDE-AND-CONQUER

General Method

Divide and conquer is a design strategy which is well known to breaking down efficiency barriers. When the method applies, it often leads to a large improvement in time complexity. Divide-and-conquer algorithms work according to the following general plan.

1. **Divide**: Divide the problem into a number of smaller sub-problems ideally of about the same size.

2. **Conquer**: The smaller sub-problems are solved, typically recursively. If the sub-problem sizes are small enough, just solve the sub-problems in a straight forward manner.

3. **Combine**: If necessary, the solution obtained the smaller problems are connected to get the solution to the original problem. The following figure shows

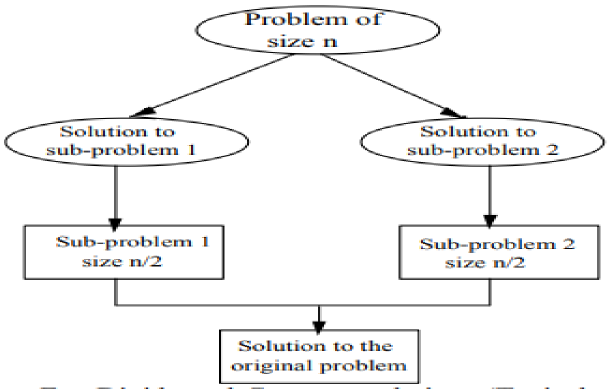

Problem of size n
Solution to sub-problem 1
Solution to sub-problem 2
Sub-problem 1 size n/2
Sub-problem 2 size n/2
Solution to the original problem

Fig: Divide-and-Conquer technique (Typical case).

Given a function to compute on *n* inputs the divide-and-conquer strategy suggests splitting the inputs into *k* distinct subsets, 1< *k* < *n*, yielding k sub problems. These sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole. If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied. Often the sub problems resulting from a divide-and conquer design are of the same type as the original problem. For those cases the reapplication of the divide-and-conquer principle is naturally expressed by a recursive algorithm.

Example : Detecting Counterfeit Coin

A bag with n 16 coins and one of these coins may be counterfeit. Counterfeit coins are lighter than genuine ones. The task is to determine whether the bag contains a counterfeit coin. It has a machine that compares the weights of two sets of coins and tells which set is lighter or whether both sets have the same weight.

We can compare the weights of coins 1 and 2. If coin 1 is lighter than coin 2, then coin 1 is counterfeit and we are done with our task. If coin 2 is lighter than coin 1, then coin 2 is counterfeit. If both coins have the same weight, we compare coins 3 and 4. Proceeding in the way, we can determine whether the bag contains a counterfeit coin by making at most eight weight comparisons. This process also identifies the counterfeit coin.

Another approach is to use the divide-and-conquer methodology. Suppose that our 16-coin instance is considered a large instance. In step 1, we divide the original instance into two or more smaller instances. Let us divide the 16-coin instance into two 8-coin instances by arbitrarily selecting 8 coins for the first instance (say A) and the remaining 8 coins eight coins for the second instance B. In step 2 we need to determine whether A or B has a counterfeit coin. For this step we use our machine to compare the weights of the coin sets A and B. If both sets have the same weight, a counterfeit coin is not present in the 16-coin set. If A and B have different weights, a counterfeit coin is present and it is in the lighter set.

To be more precise, suppose we consider the divide-and-conquer strategy when it splits the input into two sub problems of the same kind as the original problem. This splitting is typical of many of the problems we examine here. We can write a control abstraction that mirrors the way an algorithm based on divide-and-conquer will look .By a *control abstraction* we mean a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined. *DAndC* (algorithm below) is initially invoked as *DAndC* (*P*),where *P* is the problem to be solved.

```
Type DAndC(P)
{
        if Small(P)
                return S(P);
        else
        {
        divide P into smallerinstancesPi,P2,..P$_k$,  k ≥ 1;

        Apply DAndC to each of these sub problems;

        return Combine(DAndC(P$_1$),DAndC(P$_2$), .... , DAndC(P$_k$));

        }

}
```

Small(P) is a Boolean-valued function that determines whether the input Size is small enough that the answer can be computed without splitting. If this is so, the function S is invoked. Otherwise the problem *P* is divided into smaller sub problems. These sub problems P_1, P_2 .... P_k are solved by recursive applications of DAndC. Combine is a function that determines the Solution to P using the solutions to the *k* sub problems. If the size of *P* is *n* and the sizes of the *k* sub problems are n_1, n_2, ..., n_k respectively, then the computing time of DAndC is described by the recurrence relation,

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \cdots + T(n_k) \quad + \quad f(n) & \text{otherwise} \end{cases}$$

Where, ***T(n)*** is the time for DAndC on any input of size *n*

g(n) is the time to compute the answer directly for small inputs.

The function ***f(n)*** is the time for dividing *P* and combining the solutions to sub problems.

For divide-and-conquer-based algorithms that produce sub problems of the same type as the original problem, it is very natural to first describe such algorithms using recursion.

The complexity of many divide-and-conquer algorithms is given by recurrences of the form,

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

Where *a* and *b* are known constants. We assume that *T(1)*is known and *n* is a power of *b* (i.e. , $n = b^k$). One of the methods for solving any such recurrence relation is called the substitution method.

<u>Binary Search</u>

Binary search is an efficient searching technique that works with only sorted lists. So the list must be sorted before using the binary search method. Binary search is based on divide-and-conquer technique.

The method starts with looking at the middle element of the list. If it matches with the key element, then search is complete. Otherwise, the key element may be in the first half or second half of the list. If the key element is less than the middle element, then the search continues with the first half of the list. If the key element is greater than the middle element, then the search continues with the second half of the list. This process continues until the key element is found or the search fails indicating that the key is not there in the list.

Let a_i, 1<i <n, be a list of elements that are sorted in non-decreasing order. Consider the problem of determining whether a given element *x* is present in the list. If *x* is present, we are to determine a value j such that $a_j = x$. If *x* is not in the list, then *j* is to be set to zero. Let $P = (n, a_i \,..\, - a_l ,x)$ denote an arbitrary instance of this search problem (*n* is the number of elements in the list, $a_i, \dots, a_l$ is the list of elements, and *x* is the element searched for).

Divide-and-conquer can be used to solve this problem. Let Small (P) be true if n = 1. In this case, S(P)will take the value *i* if $x = a_i$, otherwise it will take the value 0. If *P* has more than one element, it can be divided (or reduced) into a new sub problem as follows. Pick an index q (in the range $[i, l]$ and compare *x* with a_q. If q is always chosen such that a_q is the middle element (that is, $q = \lfloor (n+1)/2 \rfloor$), then the resulting search algorithm is known as binary search. There are three possibilities:

(1) $x = a_q$: In this case the problem *P* is immediately solved.
(2) $x < a_q$: In this case *x* has to be searched for only in the sub list $a_2, a_{j+i}, \dots, a_{q-1}$. Therefore, *P* reduces to $(q - i, a_i, \dots, a_{q-1,x})$.
(3) $x > a_q$: In this case the sub list to be searched is $a_{q+1}, \dots, al, x)$.

<u>Example 1:</u>

Consider the list of elements: -4, -1, 0, 5, 10, 18, 32, 33, 98, 147, 154, 198, 250, 500. Trace the binary search algorithm searching for the element -1.

Sol: The given list of elements are:

Low														High
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

Searching key '-1': Here the key to search is '-1'

First calculate mid;

Mid = (low + high)/2

= (0 +14) /2 =7

Low							Mid							High
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

← First Half → ← Second Half →

Here, the search key -1 is less than the middle element (32) in the list. So the search process continues with the first half of the list.

Low						High								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

Now mid = (0+6)/2
=3.

Low			Mid			High								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

← First Half → ←Second Half→

The search key '-1' is less than the middle element (5) in the list. So the search process continues with the first half of the list.

Low		High												
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

Now mid= (0+2)/2
=1

Low	Mid	High												
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

Here, the search key -1 is found at position 1.

Example 2

List of elements are,

-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151

placed in a[l:14], and simulate the steps that *BinSearch* goes through as it searches for different values of x. Only the variables *low, high*, and *mid* need to be traced as we simulate the algorithm.

We try the following values for x: 151, -14, and 9 for two successful searches and one unsuccessful search. The following entry shows the traces of *BinSearch* on these three inputs.

$x = 151$	*low*	*high*	*mid*
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			found

$x = -14$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	not found

$x = 9$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	4	6	5
			found

Advantages of Binary Search

The main advantage of binary search is that it is faster than sequential (linear) search. Because it takes fewer comparisons, to determine whether the given key is in the list, then the linear search method.

Disadvantages of Binary Search

The disadvantage of binary search is that can be applied to only a sorted list of elements. The binary search is unsuccessful if the list is unsorted.

Efficiency of Binary Search

To evaluate binary search, count the number of comparisons in the best case, average case, and worst case.

Best Case: The best case occurs if the middle element happens to be the key element. Then only one comparison is needed to find it. Thus the efficiency of binary search is O(1).

Ex: Let the given list is: 1, 5, 10, 11, 12.

Low		Mid		High
1	5	$\boxed{10}$	11	12

Let key = 10.

Since the key is the middle element and is found at our first attempt.

Worst Case: Assume that in worst case, the key element is not there in the list. So the process of divides the list in half continues until there is only one item left to check.

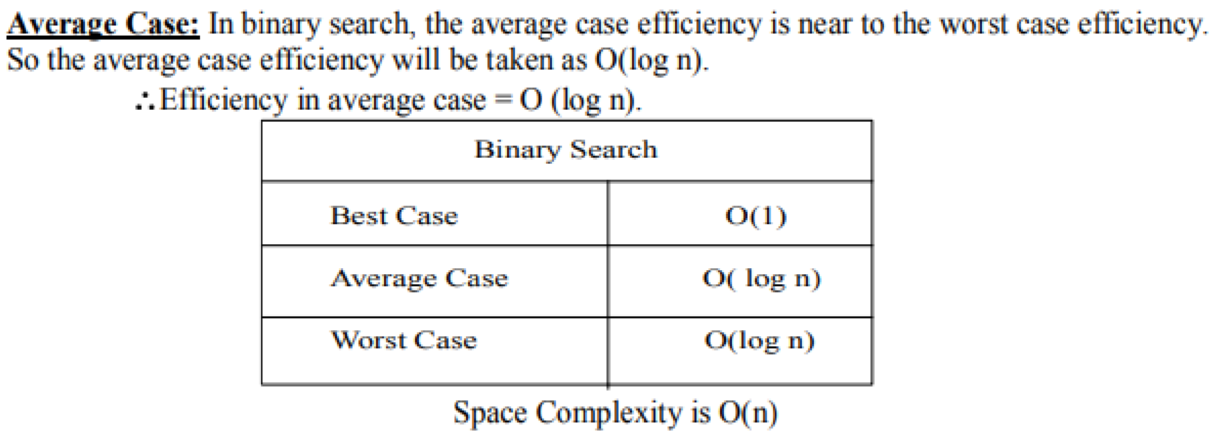
Items left to search	*Comparisons so far*
16	0
8	1
4	2
2	3
1	4

For a list of size 16, there are 4 comparisons to reach a list of size one, given that there is one comparison for each division, and each division splits the list size in half.

In general, if n is the size of the list and c is the number of comparisons, then

$$C = \log_2 n$$

$\therefore$ Eficiency in worst case = O(log n)



Average Case: In binary search, the average case efficiency is near to the worst case efficiency. So the average case efficiency will be taken as O(log n).

∴Efficiency in average case = O (log n).

Binary Search	
Best Case	O(1)
Average Case	O(log n)
Worst Case	O(log n)

Space Complexity is O(n)

Program - Recursive Binary Search

```
int BinSrch ( Type a[], int i, int l, Type x)
          // Given an array a[i :l] of elements in non-decreasing
          // order, 1<=i <=l, determine whether x is present, and
          // if so, return j such that x = = a[j]; else return 0.
{
          if (l = = i)
          {// if Small(P)
                    if (x = a[i]) return i;
                              else return 0;
          }
          else
           {// ReduceP into a smaller sub-problem.
                    Int  mid = (i+l)/2;
                     if (x = =  a[mid]) return mid;
                    else if (x < a[mid])  return BinSrch(a, i, mid-1,x);
                     else return BinSrch(a,mid+1,l,x);
          }

}
```

Program – Iterative Binary Search

```
int BinSrch ( Type a[], int n, Type x)
          // Given an array a[i :n] of elements in non-decreasing
          // order, n>=0, determine whether x is present, and
          // if so, return j such that x = = a[j]; else return 0.
{
          int  low = 1, high = n;
          while (low <= high) {
          {
                    int  mid = (low + high)/2;
                    if (x < a[mid])  high = mid - 1;
                    else if (x  > a[mid])  low =mid + 1;
                    else return (mid);
           }
           return 0;
}
```

Finding the Maximum and Minimum

The problem is to find the maximum and minimum items in a set of n elements. In analyzing the time complexity of this algorithm, concentrate on the number of element comparisons. The justification for this is that, the frequency count for other operations in this algorithm is of the same order as that for element comparisons. More importantly, when the elements in a[1:n] are polynomials, vectors, very large numbers, or strings of characters, the cost of an element comparison is much higher than the cost of the other operation.

Straight forward Minimum and Maximum Algorithm

```
        void StraightMaxMin(Type a[], int n, Type & max, Type & min)
        // Set max to the maximum and min to the minimum of a[l:n]
        {
                max = min = a[l];
                for( int i = 2; i <= n; i++)
                {
                        if (a[i]>=  max) max =a[i];
                        if (a[i]<= min) min =a[i];
                }
}
```

StraightMaxMin requires *2(n - 1)* element comparisons in the best, average, and worst cases. An immediate improvement is possible by realizing that the comparison *a[i] < min* is necessary only when *a[i] > max* is false. Hence we can replace the contents of the **for** loop by

```
        if( a[i] > max max = a[i];

        else if ( a[i] < min) min = a[i];
```

Now the best case occurs when the elements are in increasing order. The number of element comparisons is *n - 1*. The worst case occurs when the elements are in decreasing order. In this case the number of element comparison is s *2(n - 1)*.The average number of element comparisons is less than *2(n - 1)*. On the average, *a[i]* is greater than *max* half the time, and so the average number of comparisons is *3n/2 – 1*.

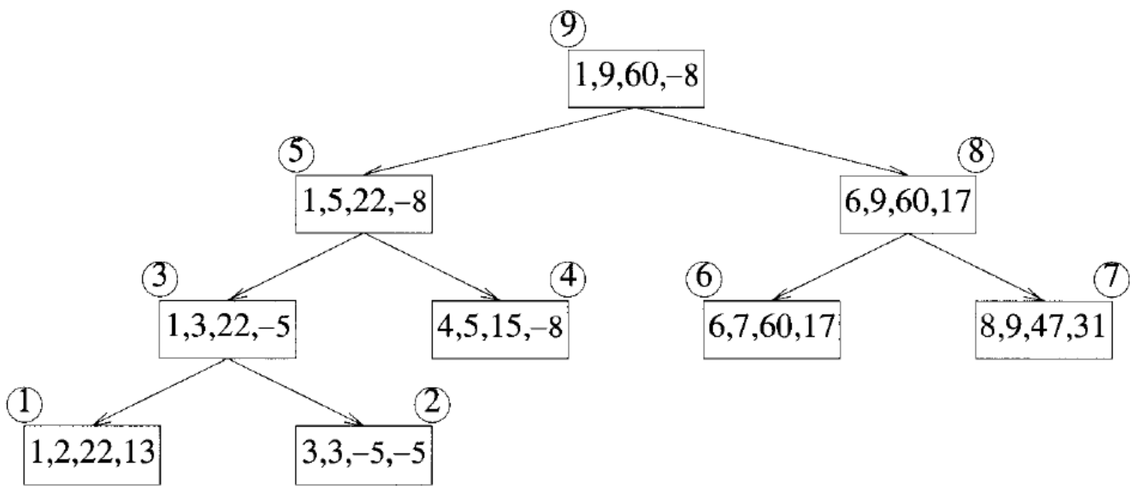
A divide-and-conquer algorithm for this problem would proceed as follows: Let *P = (n,a[i], ..., a[j])* denote an arbitrary instance of the problem. Here *n* is the number of elements in the list a[i], ..., a[j] and we are interested in finding the maximum and minimum of this list. Let *Small(P)* be true when n $\leq$ 2. In this case, the maximum and minimum are a[i] if n = 1. If n = 2, the problem can be solved by making one comparison.

If the list has more than two elements, *P* has to be divided into smaller instances. For example, we might divide *P* into the two instances $P_1 = (\lfloor n/2 \rfloor, a[l], ..., a[\lfloor n/2 \rfloor])$ and $P_2 = (n - \lfloor n/2 \rfloor, a[\lfloor n/2 \rfloor + 1], ..., a[n])$. After having divided *P* into two smaller sub problems it can solve by recursively invoking the same divide-and-conquer algorithm. If MAX(P) and MIN(P) are the maximum and minimum of the elements in P, then MAX(P) is the larger of MAX(P_1) and MAX(P_2). Also, MIN(P) is the smaller of MIN(P_1)and MIN(P_2).

The following algorithm is recursively finding the maximum and minimum.

```
void MaxMin (int i, int j, Type & max, Type & min)
    // a[1:n] is a global array. Parameters i and j are integers,
    // 1 <= i <= j <=n. The effect is to set max and min to the
     // largest and smallest values in a[i :j],respectively.
     {
    if (i = = j) max = min = a[i]; // Small(P)
    else if (i = = j - 1) { // Another case of Small(P)
            if (a[i] < a[j])  { max = a[j]; min =a[i]; }
            else  {  max = a[i]; min:=a[j];  }
                        }
     else { // if Pis not small, divide P into sub-problems.
     // Find where to split the set.
    int mid = (i+j)/2; Tpe max1, min1;
     // Solve the sub-problems.
     MaxMin (i, mid, max, min);
     MaxMin (mid+l, j, maxl, minl);
     // Combine the solutions.
     if (max < max1) max = maxl;
    if (min  > min1)  min = mini;
          }
    }
```

<u>Trees of Recursive Calls of *MaxMin*</u>



a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	22	13	−5	−8	15	60	17	31	47

Merge Sort

Merge sort is based on divide-and-conquer technique. Merge sort method is a two phase process,

1. Dividing
2. Merging

Dividing Phase: During the dividing phase, each time the given list of elements is divided into two parts. This division process continues until the list is small enough to divide.

Merging Phase: Merging is the process of combining two sorted lists, so that, the resultant list is also the sorted one. Suppose A is a sorted list with *n1* element and B is a sorted list with *n2* elements. The operation that combines the elements of A and B into a single sorted list C with *n=n1 + n2,* elements is called merging.

A sorting algorithm that has the nice property that in the worst case its complexity is O (n log n). This algorithm is called merge sort. We assume throughout that the elements are to be sorted in non-decreasing order. Given a sequence of n elements (also called keys) a[1], ..., a[n], the general idea is to imagine them split into two sets a[1], ... , a[$\lfloor n/2 \rfloor$] and a[$\lfloor n/2 \rfloor$+1], ... , a[n]. Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of *n* elements. Thus we have another ideal Example of the divide-and-conquer strategy in which the splitting is into two equal-sized sets and the combining operation is the merging of two sorted sets into one.

Merge Sort - Algorithm

```
void MergeSort (int low, int  high)
 // a[low :high] is a global array to be sorted.
// Small(P) is true if there is only one element
// to sort. In this case the list is already sorted.
{
         if (low < high) { //If there are more than one element
                              // Divide P into sub-problems.
                            // Find where to split the set.
        int  mid = (low + high)/2;
                           // Solve the sub-problems.
         MergeSort (low, mid);
         MergeSort (mid + 1, high);
                            // Combinethe solutions.
         Merge(low, mid, high);

                           }

        }
```

<u>Merging two Sorted Sub-arrays using Auxiliary Storage</u>

```
void Merge( int low, int mid, int high)
    // a[low :high] is a global array containing two sorted
    // subsets in a [low :mid] and in a[mid+ 1: high]. The goal
    //is to merge these two sets into a single set residing in a[low : high].
    // b[ ] is an auxiliary global array.
{
          int h =low; i = low; j = mid + 1, k;
          while ((h <= mid) &&  (j <=  high)) {
                    if (a[h] <= a[j]) {b[i] = =a[h]; h = h + 1; }
                    else {b[i] = a[j]j;  j++;} i++;

                                                  } //while

          if (h > mid) for( k = j; k <= high; k++) {
                              b[i] =a[k]; i++;
                                        }
          else for( k = h; k <= mid; k++) {
                              b[i] =a[k]; i++;
                                        }
          for( k = low; k <= high; k++ ) a[k] = b[k];
}
```

<u>Example:</u>

Consider the array of ten elements $a[1:10] = (310, 285, 179,$ $652, 351, 423, 861, 254, 450, 520)$. Algorithm MergeSort begins by splitting $a[\,]$ into two subarrays each of size five ($a[1:5]$ and $a[6:10]$). The elements in $a[1:5]$ are then split into two subarrays of size three ($a[1:3]$) and two ($a[4:5]$). Then the items in $a[1:3]$ are split into subarrays of size two ($a[1:2]$) and one ($a[3:3]$). The two values in $a[1:2]$ are split a final time into one-element subarrays, and now the merging begins. Note that no movement of data has yet taken place. A record of the subarrays is implicitly maintained by the recursive mechanism. Pictorially the file can now be viewed as

$$(310 \mid 285 \mid 179 \mid 652, 351 \mid 423, 861, 254, 450, 520)$$

where vertical bars indicate the boundaries of subarrays. Elements $a[1]$ and $a[2]$ are merged to yield

$$(285, 310 \mid 179 \mid 652, 351 \mid 423, 861, 254, 450, 520)$$

Then $a[3]$ is merged with $a[1:2]$ and

$$(179,\ 285,\ 310 \mid 652,\ 351 \mid 423,\ 861,\ 254,\ 450,\ 520)$$

is produced. Next, elements $a[4]$ and $a[5]$ are merged:

$$(179,\ 285,\ 310 \mid 351,\ 652 \mid 423,\ 861,\ 254,\ 450,\ 520)$$

and then $a[1:3]$ and $a[4:5]$:

$$(179,\ 285,\ 310,\ 351,\ 652 \mid 423,\ 861,\ 254,\ 450,\ 520)$$

At this point the algorithm has returned to the first invocation of MergeSort and is about to process the second recursive call. Repeated recursive calls are invoked producing the following subarrays:

$$(179,\ 285,\ 310,\ 351,\ 652 \mid 423 \mid 861 \mid 254 \mid 450,\ 520)$$

Elements $a[6]$ and $a[7]$ are merged. Then $a[8]$ is merged with $a[6:7]$:

$$(179,\ 285,\ 310,\ 351,\ 652 \mid 254,\ 423,\ 861 \mid 450,\ 520)$$

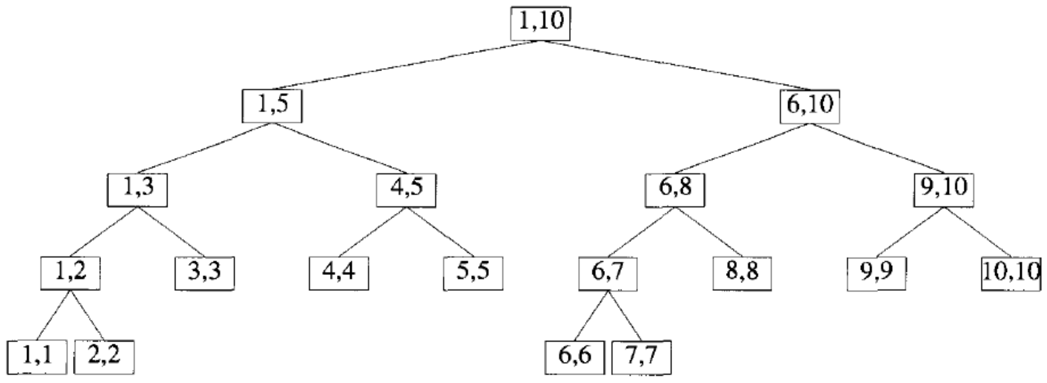
Next $a[9]$ and $a[10]$ are merged, and then $a[6:8]$ and $a[9:10]$:

$$(179,\ 285,\ 310,\ 351,\ 652 \mid 254,\ 423,\ 450,\ 520,\ 861)$$

At this point there are two sorted subarrays and the final merge produces the fully sorted result

$$(179,\ 254,\ 285,\ 310,\ 351,\ 423,\ 450,\ 520,\ 652,\ 861)$$

Tree of Calls of MergeSort(1l0,)



Quick Sort

The divide-and-conquer approach can be used to arrive at an efficient sorting method different from merge sort. In merge sort, the file $a[1:n]$ was divided

at its midpoint into subarrays which were independently sorted and later merged. In quicksort, the division into two subarrays is made so that the sorted subarrays do not need to be merged later. This is accomplished by rearranging the elements in $a[1:n]$ such that $a[i] \leq a[j]$ for all i between 1 and m and all j between $m+1$ and n for some m, $1 \leq m \leq n$. Thus, the elements in $a[1:m]$ and $a[m+1:n]$ can be independently sorted. No merge is needed. The rearrangement of the elements is accomplished by picking some element of $a[\ ]$, say $t = a[s]$, and then reordering the other elements so that all elements appearing before t in $a[1:n]$ are less than or equal to t and all elements appearing after t are greater than or equal to t. This rearranging is referred to as *partitioning*.

The function *Partition* in the algorithm accomplishes an in-place partitioning of the elements of a[m:p - 1]. It is assumed that a[p] ≥ a[m] and that a[m] is the partitioning element. If m = 1 and p - 1 = n, then a[n + 1] must be defined and must be greater than or equal to all elements in a[1:n]. The assumption that a[m] is the partition element is merely for convenience; other choices for the partitioning element than the first item in the set are better in practice. The function *Interchange (a, i, j)* exchanges *a[i]* with *a[j].*

```
int Partion (Type a[], int m, int p)
 // Within a[m], a[m+1], ... , a[p-1] the elements are
 // rearranged in such a manner that if initially t = = a[m],
 // then after completion a[q] = = t for some q between m and p - 1, a[k] <= t
 //for  m <= k < q, and a[k] >= t for q <k <p. q is returned.
 {
        Type  v = a[m]; int i = m; j=p;
        do {
                do i++
                while (a[i] < v);
                do j--;
                while (a[j] > v);
                 if (i < j) Interchange (a, i, j);
        } while (i < j);
        a[m] = a[j]; a[j] =v;  return(j);
 }


 inline void Interchange(Type a[], int i, int j)
 // Exchange a[i] with a[j].
 {
        Type  p = a[i];
         a[i] = a[j;  a[j] = p;
}
```

Example:

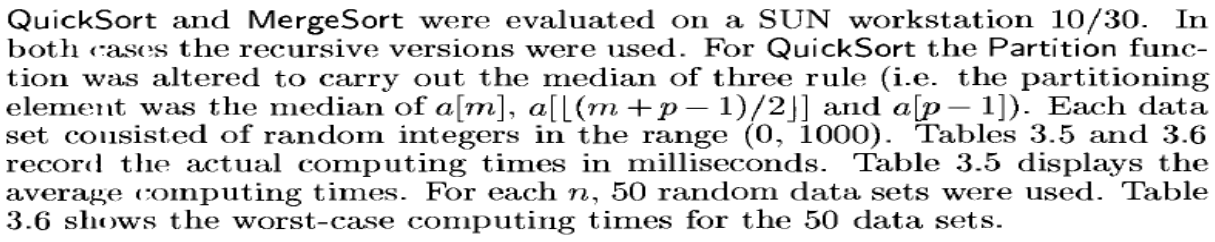
As an example of how Partition works, consider the following array of nine elements. The function is initially invoked as Partition$(a, 1, 10)$. The ends of the horizontal line indicate those elements which were interchanged to produce the next row. The element $a[1] = 65$ is the partitioning element and it is eventually (in the sixth row) determined to be the fifth smallest element of the set. Notice that the remaining elements are unsorted but partitioned about $a[5] = 65$.

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	i	p
65	70	75	80	85	60	55	50	45	$+\infty$	2	9
65	45	75	80	85	60	55	50	70	$+\infty$	3	8
65	45	50	80	85	60	55	75	70	$+\infty$	4	7
65	45	50	55	85	60	80	75	70	$+\infty$	5	6
65	45	50	55	60	85	80	75	70	$+\infty$	6	5
60	45	50	55	65	85	80	75	70	$+\infty$		

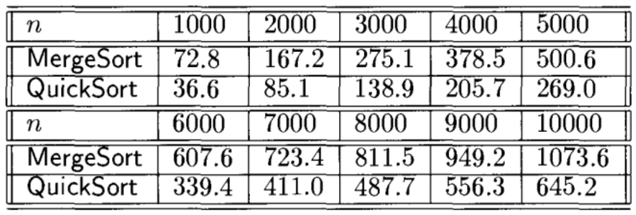
Using Hoare's clever method of partitioning a set of elements about a chosen element, we can directly devise a divide-and-conquer method for completely sorting n elements. Following a call to the function Partition, two sets S_1 and S_2 are produced. All elements in S_1 are less than or equal to the elements in S_2. Hence S_1 and S_2 can be sorted independently. Each set is sorted by reusing the function Partition. The following algorithm describes the complete process.

```
void QuickSort (int p, int q)
  // Sorts the elements alp], ..., a[q] which reside in the global
  // array a[1:n] into ascending order;  a[n+ 1] is considered to
  // be defined and must be >=  all the elements in a[l:n].
{
        if (p < q)  { // If there are more than one element
        // divide P into two sub-problems.
                int  j = Partition (a, p, q+ 1);
        // j is the position of the partitioning element.
        // Solve the sub-problems.
                QuickSort (p, j-1);
                QuickSort(j+ l, q);
        // There is no need for combining solutions.
                }
}
```

Performance Measurement

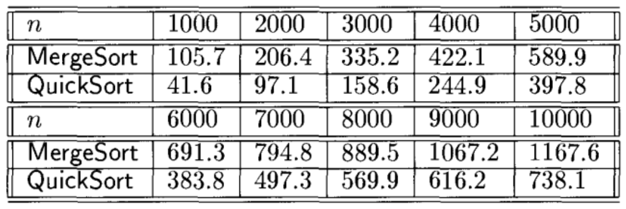


QuickSort and MergeSort were evaluated on a SUN workstation 10/30. In both cases the recursive versions were used. For QuickSort the Partition function was altered to carry out the median of three rule (i.e. the partitioning element was the median of $a[m]$, $a[\lfloor(m+p-1)/2\rfloor]$ and $a[p-1]$). Each data set consisted of random integers in the range (0, 1000). Tables 3.5 and 3.6 record the actual computing times in milliseconds. Table 3.5 displays the average computing times. For each n, 50 random data sets were used. Table 3.6 shows the worst-case computing times for the 50 data sets.



n	1000	2000	3000	4000	5000
MergeSort	72.8	167.2	275.1	378.5	500.6
QuickSort	36.6	85.1	138.9	205.7	269.0
n	6000	7000	8000	9000	10000
MergeSort	607.6	723.4	811.5	949.2	1073.6
QuickSort	339.4	411.0	487.7	556.3	645.2

Table3.5 Average computing times for two sorting algorithms on random inputs



n	1000	2000	3000	4000	5000
MergeSort	105.7	206.4	335.2	422.1	589.9
QuickSort	41.6	97.1	158.6	244.9	397.8
n	6000	7000	8000	9000	10000
MergeSort	691.3	794.8	889.5	1067.2	1167.6
QuickSort	383.8	497.3	569.9	616.2	738.1

Table3.6 Worst-case computing times for two sorting algorithms on random inputs

Scanning the tables, we immediately see that *QuickSort* is faster than *MergeSort* for all values. Even though both algorithms require *O(n log n)* time on the average, *QuickSort* usually performs well in practice.

Selection

The *Partition* algorithm can also be used to obtain an efficient Solution for the selection problem. In this problem, we are given n elements a[1 : n] and are required to determine the *k*th-smallest element. If the Partitioning element *v* is positioned at a[j], then *j-1* elements are less than or equal to a[j] and *n- j* elements are greater than or equal to a[j]. Hence if *k < j,* then the *k*th-smallest element is in a[l: j - 1]; if *k = j,* then a[j] is the *k*th-smallest element; and if *k >j,* then the *k*th-smallest element is the (k - j)th-smallest element in *a[j + 1: n]*. The resulting algorithm is function ***Selectl*** below. This function places the *k*th-smallest element into position a[k] and partitions the remaining elements so that *a[i] ≤ a[k], 1≤ i < k,* and *a[i] ≥a[k], k < i ≤ n.*

```
void Selectl (Type a[], int a, int k)
   // Selects the kth-smallest element in a[l:n] and places it in the kth position of a[ ]
   //  The remaining elements are rearranged such that  a[m] <= a[k] for 1<= m < k, and
  // a[m] >= a[k] for k <m <=n.
{
          int low =1; up = n + 1;
```

```
        a[n+ 1]= INFTY; // a[n+ 1]is set to infinity.
        do { // Each time the loop is entered,  1<= low <=k <=up <= n+ 1.
                int j :=Partition(a, low, up);
                        //j is such that a[j] is the jth-smallest value in a[ ].
                if (k = = j) return;
                else if (k < j) up =j; // j is the new upper limit.
                else low = j + 1; // j + 1is the new lower limit.
        }while(TRUE);
}
```

<u>Example</u>

The array has the nine elements 65, 70, 75, 80, 85, 60, 55, 50, and 45, with a[10] = ∞. If k = 5, then the first call of *Partition* will be sufficient since 65 is placed into a[5]. Instead, assume that we are looking for the seventh-smallest element of a, that is, k = 7. The next invocation of Partition is Partition (6, 10).

a:	(5)	(6)	(7)	(8)	(9)	(10)
	65	85	80	75	70	$+\infty$
	65	70	80	75	85	$+\infty$

This last call of Partition has uncovered the ninth-smallest element of a. The next invocation is Partition(6, 9).

a:	(5)	(6)	(7)	(8)	(9)	(10)
	65	70	80	75	85	$+\infty$
	65	70	80	75	85	$+\infty$

This time, the sixth element has been found. Since $k \neq j$, another call to Partition is made, Partition(7, 9).

a:	(5)	(6)	(7)	(8)	(9)	(10)
	65	70	80	75	85	$+\infty$
	65	70	75	80	85	$+\infty$

Now 80 is the partition value and is correctly placed at $a[8]$. However, Select1 has still not found the seventh-smallest element. It needs one more call to Partition, which is Partition(7, 8). This performs only an interchange between $a[7]$ and $a[8]$ and returns, having found the correct value. □

Strassen's Matrix Multiplication

Let A and B be two $n \times n$ matrices. The product matrix $C = AB$ is also an $n \times n$ matrix whose i, jth element is formed by taking the elements in the ith row of A and the jth column of B and multiplying them to get

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

for all i and j between 1 and n. To compute $C(i,j)$ using this formula, we need n multiplications. As the matrix C has n^2 elements, the time for the resulting matrix multiplication algorithm, which we refer to as the conventional method is $\Theta(n^3)$.

The divide-and-conquer strategy suggests another way to compute the product of two $n \times n$ matrices. For simplicity we assume that n is a power of 2, that is, that there exists a nonnegative integer k such that $n = 2^k$. In case n is not a power of two, then enough rows and columns of zeros can be added to both A and B so that the resulting dimensions are a power of two

Imagine that A and B are each partitioned into four square submatrices, each submatrix having dimensions $\frac{n}{2} \times \frac{n}{2}$. Then the product AB can be computed by using the above formula for the product of 2×2 matrices: if AB is

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \qquad (3.11)$$

then

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned} \qquad (3.12)$$

If $n = 2$, then formulas (3.11) and (3.12) are computed using a multiplication operation for the elements of A and B. These elements are typically floating point numbers. For $n > 2$, the elements of C can be computed using *matrix* multiplication and addition operations applied to matrices of size $n/2 \times n/2$. Since n is a power of 2, these matrix products can be recursively computed by the same algorithm we are using for the $n \times n$ case. This algorithm will continue applying itself to smaller-sized submatrices until n becomes suitably small ($n = 2$) so that the product is computed directly.

To compute AB using (3.12), we need to perform eight multiplications of $n/2 \times n/2$ matrices and four additions of $n/2 \times n/2$ matrices. Since two $n/2 \times n/2$ matrices can be added in time cn^2 for some constant c, the overall computing time $T(n)$ of the resulting divide-and-conquer algorithm is given by the recurrence

$$T(n) = \begin{cases} b & n \le 2 \\ 8T(n/2) + cn^2 & n > 2 \end{cases}$$

where b and c are constants.

This recurrence can be solved in the same way as earlier recurrences to obtain $T(n) = O(n^3)$. Hence no improvement over the conventional method has been made. Since matrix multiplications are more expensive than matrix additions ($O(n^3)$ versus $O(n^2)$), we can attempt to reformulate the equations for C_{ij} so as to have fewer multiplications and possibly more additions. Volker Strassen has discovered a way to compute the C_{ij}'s of (3.12) using only 7 multiplications and 18 additions or subtractions. His method involves first computing the seven $n/2 \times n/2$ matrices P, Q, R, S, T, U, and V as in (3.13). Then the C_{ij}'s are computed using the formulas in (3.14). As can be seen, P, Q, R, S, T, U, and V can be computed using 7 matrix multiplications and 10 matrix additions or subtractions. The C_{ij}'s require an additional 8 additions or subtractions.

$$\begin{array}{lcl} P & = & (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q & = & (A_{21} + A_{22})B_{11} \\ R & = & A_{11}(B_{12} - B_{22}) \\ S & = & A_{22}(B_{21} - B_{11}) \\ T & = & (A_{11} + A_{12})B_{22} \\ U & = & (A_{21} - A_{11})(B_{11} + B_{12}) \\ V & = & (A_{12} - A_{22})(B_{21} + B_{22}) \end{array} \tag{3.13}$$

$$\begin{array}{lcl} C_{11} & = & P + S - T + V \\ C_{12} & = & R + T \\ C_{21} & = & Q + S \\ C_{22} & = & P + R - Q + U \end{array} \tag{3.14}$$

The resulting recurrence relation for $T(n)$ is

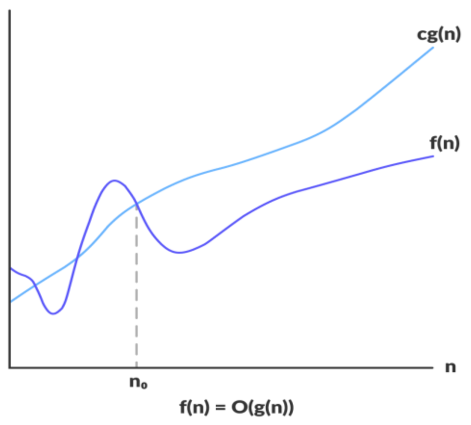
$$T(n) = \begin{cases} b & n \leq 2 \\ 7T(n/2) + an^2 & n > 2 \end{cases} \tag{3.15}$$

Where *a* and *b* are constants.

ooooo-ooooo-ooooo

Big O notation

Big O notation tells the number of operations an algorithm will make. It gets its name from the literal "Big O" in front of the estimated number of operations. Big-O notation represents the *upper bound* of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.



O(g(n)) = { f(n): there exist positive constants c and n0

such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0 }

This expression can be described as a function f(n) belongs to the set O(g(n)) if there exists a positive constant c such that it lies between 0 and cg(n), for sufficiently large n.

For any value of n, the running time of an algorithm does not cross the time provided by O(g(n)).

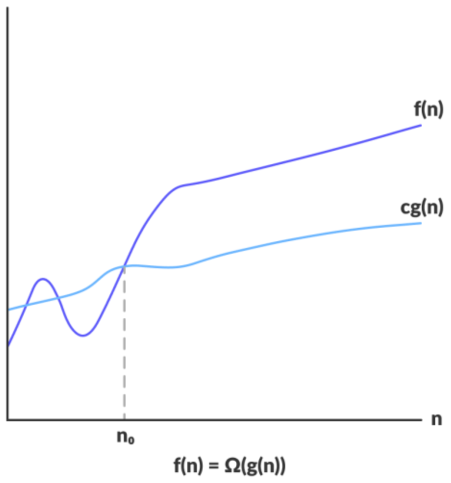
Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm in the worst-case scenario.

Here are some common algorithms and their run times in Big O notation:

BIG O NOTATION	EXAMPLE ALGORITHM
O(log n)	Binary search
O(n)	Simple search
O(n * log n)	Quick sort
O(n2)	Selection sort
O(n!)	Travelling salesperson

Omega Notation (Ω-notation)

Omega notation represents the *lower bound* of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.



Ω(g(n)) = { f(n): there exist positive constants c and n0

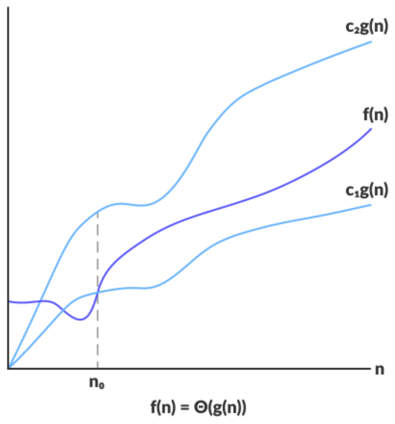
such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n0 }

This expression can be described as a function f(n) belongs to the set Ω(g(n)) if there exists a positive constant c such that it lies above cg(n), for sufficiently large n.

For any value of n, the minimum time required by the algorithm is given by Omega Ω(g(n))

Theta Notation (Θ-notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.



For a function g(n), Θ(g(n)) is given by the relation:

Θ(g(n)) = { f(n): there exist positive constants c1, c2 and n0

such that $0 \le c1g(n) \le f(n) \le c2g(n)$ for all $n \ge n0$ }

This expression can be described as a function f(n) belongs to the set Θ(g(n)) if there exist positive constants c1 and c2 such that it can be sandwiched between c1g(n) and c2g(n), for sufficiently large n.

If a function f(n) lies anywhere in between c1g(n) and c2g(n) for all $n \ge n0$, then f(n) is said to be asymptotically tight bound.