MODULE—2
DIVIDE AND CONQUER

2.1 Divide and conquer General Method

The Divide-and-conquer strategy suggests splitting the inputs of size ‘n’ into ‘k’ distinct subsets such that $1<k \leq n$, producing k sub problems.

These sub problems must be solved & then a method to be found to combine the solutions of sub problems (sub solutions) to produce the solution to the original problem of size ‘n’. If the sub problems are relatively large then divide & conquer strategy must be reapplied. For the reapplication sub problems the divide & conquer strategy will be expressed as recursive algorithm.

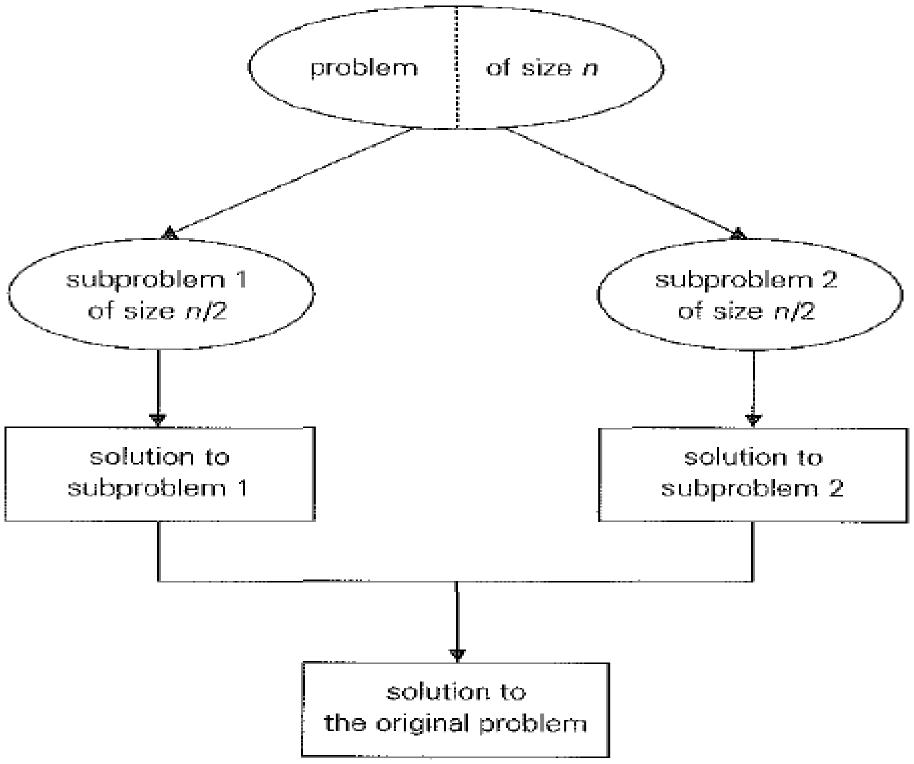


Figure 2.1: Divide and conquer

Thus, the general plan of divide and conquer strategy can be represented as follows:

1. **DIVIDE:** A problems instance is divided into several sub problem instances of the same problem, ideally of about the same size.
2. **RECUR:** solve the sub problems recursively
3. **CONQUER :** If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance

- Divide-and-Conquer (DaC) is probably the best-known general algorithm design technique.
- Given a function to compute on n input the DaC approach suggests splitting the inputs into k distinct subsets,1< k < n, yielding k subproblems

- These sub problems must be solved and then a method must be found to combine solutions into a solution of the whole.
- If the sub problems are relatively large then the divide and conquer approach can possibly be reapplied
- Often the sub problems resulting from our divide and conquer design are of the same type as the original problem.
- Further those cases the reapplication of the divide and conquer principle is naturally expressed by a recursive algorithm
- The smaller and smaller sub problems of the same kind are generated until eventually sub problems that are small enough to be solved without splitting are produced

Divide and conqueror strategy splits the input into two sub problems of the same kind as the original problem. Consider the following algorithm (DAndC) which is invoked for problem p to be solved.

Small (P) is a Boolean-valued function that determines whether the input size is small enough that the answer can be computed without splitting.

```
Algorithm DAndC($P$)
{
    if Small($P$) then return S($P$);
    else
    {
        divide $P$ into smaller instances $P_1, P_2, \ldots, P_k$, $k \geq 1$;
        Apply DAndC to each of these subproblems;
        return Combine(DAndC($P_1$),DAndC($P_2$),...,DAndC($P_k$));
    }
}
```

If the size of p is n and the sizes of k sub problems are n_1, n_2,….., n_k then the computing time of (DAndC) is described by the recurrence relation. Divide and conquer (DAndC) is described by the recurrence relation given below which is used to know the computing time.

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

T(n) - The time for DAnd C on any input of size n

g(n) -Time to compute the answer directly for small inputs.

f(n)- Time for dividing P and combining the solutions to sub problems.

T(n $_{1...k}$) – Time taken for instances of sub problems.

Solve the Problem:

Consider the case in which a=2 and b=2. Let T(1)=2 and f(n)= n .

Solution : refer to material sent

2.2 Divide-and-conquer recurrence equation

In the most typical case of divide-and-conquer , a problem's instance of size *n* is divided into two instances of size *n/*2.

i, e $T(n) = T(n/2) + T(n/2)+ f(n)$

$= 2\, T(n/2) + f(n)$

More generally, an instance of size *n* can be divided into *b* instances of size *n/b*, with *a* of them to be solved. (Here, *a* and *b* are constants; $a \geq 1$ and $b > 1$.)
Assuming that size *n* is a power of *b* to simplify analysis; we get the following recurrence for the running time *T (n)*:

$$\boxed{T(n) = a\,T(n/b) + f(n)} \quad \text{..... 2.1}$$

Where **f *(n)*** is a function that accounts for the time spent on dividing an instance of size *n* into instances of size *n/b* and combining their solutions.
By recursively solving equation 2.1 , we get

$$T(n)= n^{\log_b a}\,[T(1) + u(n)]$$

Where,

$$u(n)= \sum^{k}_{j=1} h(b^{j})$$

$$h(n)= f(n)/ n^{\log_b a}$$

2.3 Binary Search

Let a_i $(1\leq i\leq n)$ is a list of elements stored in non decreasing order. The searching problem is to determine whether a given element is present in the list. If key x is present, we have to determine the index value j such that a[j] =x. If x is not in the list j is set to be zero.

Let P= (n, a_i, . . . a_l, x) denotes an instance of binary search problem. Divide & Conquer can be used to solve this binary search problem. Let Small (P) be true if n=1. S(P) will take the value i if x=a_i , otherwise it will take 0.

If P has more than one element it can be divided into new sub-problems as follows:

Take an index q within the range [i, l] & compare x with aq.

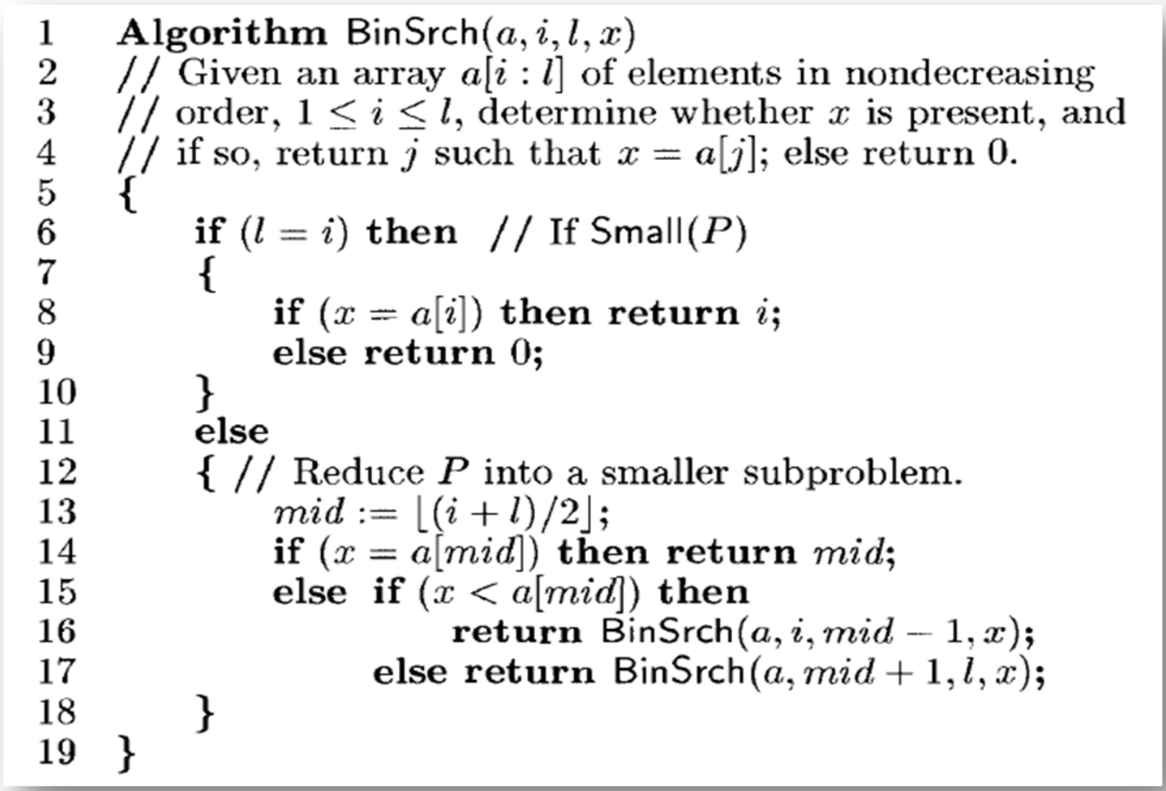
There are three possibilities.

i. If x= a_q the problem is immediately solved (Successful search)

ii. If x< a_q , key x is to be searched only in sub list a_i, a_{i+1}, . . ., a_{q-1}.

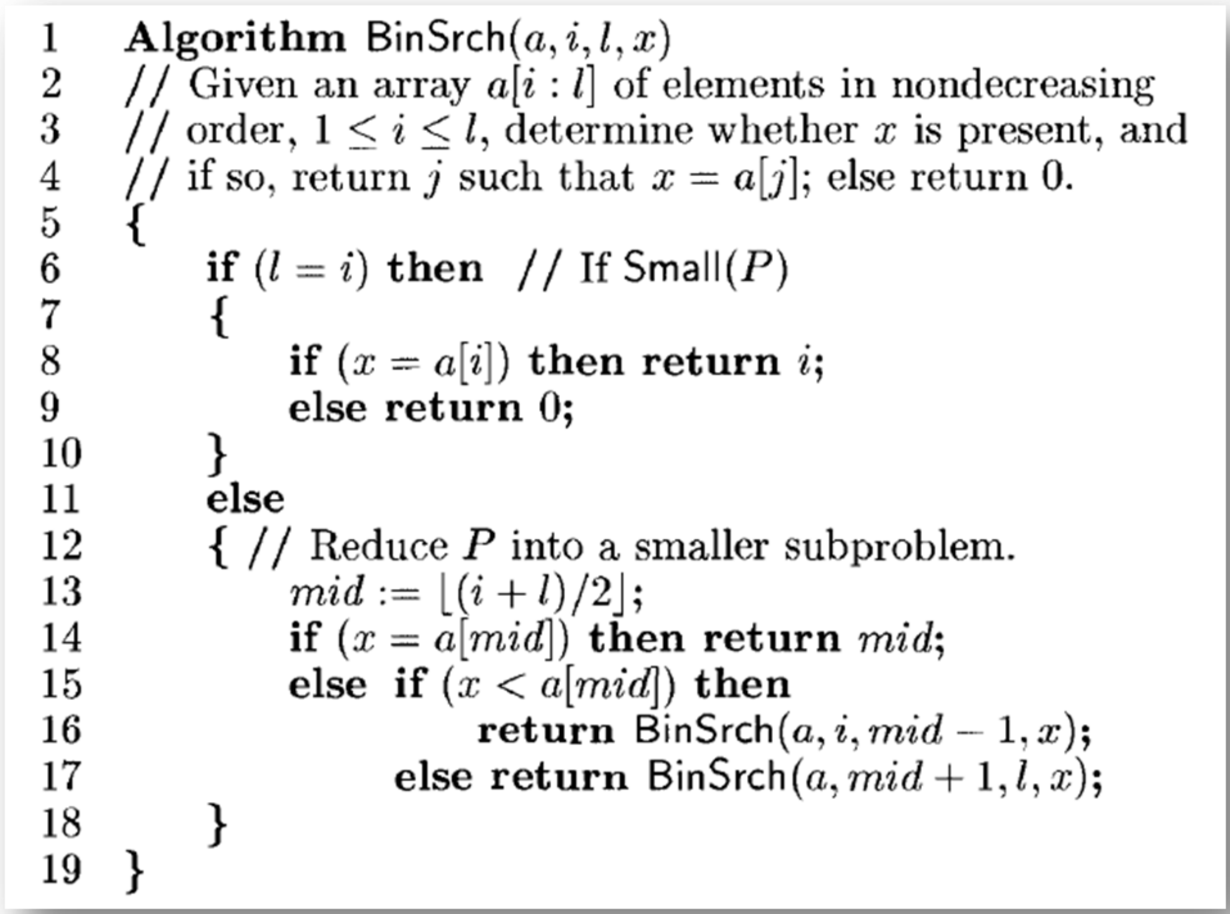
iii. If x >a_q, key x is to be searched only in sub list a_{q+1}, a_{q+2},....a_l. If q is chosen such that a_q is the middle element i.e. q=(n+1)/2, then the resulting searching algorithm is known as Binary Search algorithm.

Non- Recursive Binary Search



```
Algorithm BinSrch($a, i, l, x$)
// Given an array $a[i : l]$ of elements in nondecreasing
// order, $1 \leq i \leq l$, determine whether $x$ is present, and
// if so, return $j$ such that $x = a[j]$; else return 0.
{
    if ($l = i$) then   // If Small($P$)
    {
        if ($x = a[i]$) then return $i$;
        else return 0;
    }
    else
    { // Reduce $P$ into a smaller subproblem.
        $mid := \lfloor (i + l)/2 \rfloor$;
        if ($x = a[mid]$) then return $mid$;
        else if ($x < a[mid]$) then
                    return BinSrch($a, i, mid - 1, x$);
                else return BinSrch($a, mid + 1, l, x$);
    }
}
```

Recursive Binary search Algorithm:



```
Algorithm BinSrch($a, i, l, x$)
// Given an array $a[i : l]$ of elements in nondecreasing
// order, $1 \leq i \leq l$, determine whether $x$ is present, and
// if so, return $j$ such that $x = a[j]$; else return 0.
{
    if ($l = i$) then   // If Small($P$)
    {
        if ($x = a[i]$) then return $i$;
        else return 0;
    }
    else
    { // Reduce $P$ into a smaller subproblem.
        $mid := \lfloor (i + l)/2 \rfloor$;
        if ($x = a[mid]$) then return $mid$;
        else if ($x < a[mid]$) then
                    return BinSrch($a, i, mid - 1, x$);
              else return BinSrch($a, mid + 1, l, x$);
    }
}
```

Analysis:-

We do the analysis with frequency count(Key operation count) and space required for the algorithm. In binary search the space is required to store n elements of the array and to store the variables low, high, mid and x i.e. n+4 locations

To find the time complexity of the algorithm, as the comparison count depends on the specifics of the input, so we need to analyze best case, average case & worst case efficiencies separately. We assume only one comparison is needed to determine which of the three possibilities of if condition in the algorithm holds. Let us take an array with 14 elements.

Index pos	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Elements	-10	-6	-3	-1	0	2	4	9	12	15	18	19	22	30
comparisons	3	4	2	4	3	4	1	4	3	4	2	4	3	4

From the above table we can conclude that, no element requires more the 4 comparisons. The average number of comparisons required is (sum of count of comparisons/number of elements) i.e 45/14=3.21 comparisons per successful search on average.

There are 15 possible ways that an unsuccessful search may terminate depending on the value of x. if x < a[1], the algorithm requires three comparisons to determine x is not present. For all the remaining possibilities the algorithm requires 4 element comparisons. Thus the average number of comparisons for an unsuccessful search is (3+14*4)/15=59/15=3.93.

To derive the generalized formula and for better understanding of algorithm is to consider the sequence of values for mid that are produced by **BinarySearch** for all possiblevalues of x. these possible values can described using binary decision tree in which the value in each node is the value of *mid*. The below figure is the decision tree for n=14.

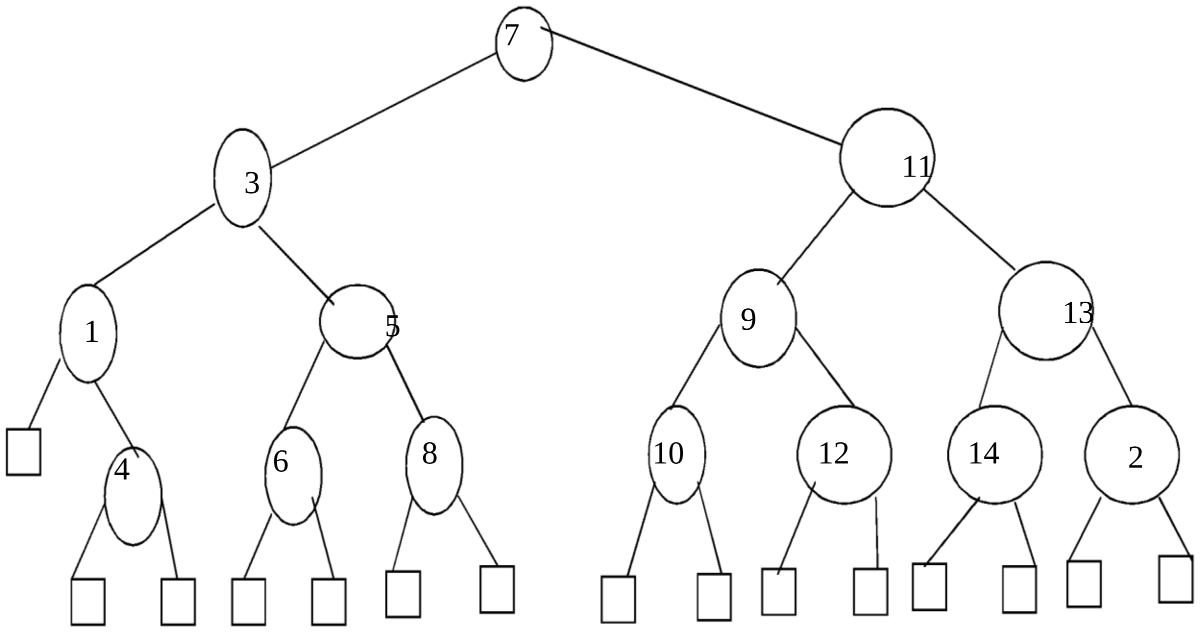


Figure 2.1: Decision tree

In the above decision tree, each path through the tree represents a sequence of comparisons in the binary search method. If x is present, then algorithm will end at one of the circular node (internal nodes) that lists the index into the array where x was found. If x is not present, the algorithm will terminate at one of the square nodes (external nodes).

Recurrence Relation

- $T(n) = \begin{cases} 1 & n = 1 \\ T\left(\frac{n}{2}\right) + 1 & n > 1 \end{cases}$

Solution

$T(n)= T(n/2)+1$

$= [T(n/4)+1]+1 = T(n/4)+2$

$=[T(n/8)+1]+2 = T(n/8) +3$

- - - - - - - - - - - - -

$=T(n/2^k)+k \qquad n =b^k , n=2^k , \log n= \log 2^k , k=\log n$

$=T(n/n)+k$

$=1+k$

$=1+\log n$

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

$T(n)= \Theta (\log n)$ [in many conventions]

Theorem 1:

If n is in the range $[2^{k-1}, 2^k]$, then **Binary Search** makes at most k element comparisons for successful search and either k-1 or k comparisons for an unsuccessful search. (i.e. The time for a successful search is O (logn) and for unsuccessful search is Θ(logn)).

Proof:

Let us consider a binary decision tree, which describes the function of Binary search on n elements. Here all successful search in figure 2.1 end at a circular node, where as all unsuccessful search will end at square node. If $2^{k-1} \leq n < 2^k$, then all circular nodes are at levels 1,2,…,k, where as all square nodes are at levels k & K+1(root node is at level 1). The number of element comparisons needed to terminate at a circular node on level i is i, where as the number of element comparisons needed to terminate at a square node at level i is only i-1, hence the proof.

The above proof will tell the worst case time complexity is O(logn) for successful search & Θ(logn) for unsuccessful search.

Overall Time complexity of binary search is

For Successful search

Best:Θ(1), Average : Θ (logn) Worst: Θ (logn)

For unsuccessful case (all same)

Best, Average, Worst: Θ (logn)

Example 1: Consider an array A of elements below for binary search
-15, -6, 0, 7, 9 , 23, 54, 82 and **Key=7**

Recursive calls: BinarySearch(a,1,12,7) for the recursive algorithm defined above BinarySearch(a, i, l, x).

Solution:

Index	1	2	3	4	5	6	7	8
Array a elements	**-15**	**-6**	**0**	**7**	**9**	**23**	**54**	**82**

- Mid =(1+8)/2=4.5= rounded to 4
- Compare key x with mid element i.e. 7=a[mid]=a[4]=7
- Key matches with the middle element of the array so the algorithm returns 4 as the index position of the element found in array a. so it is successful.
- This case is the best case also .As it takes only one comparison.

Example 2:

Consider an array A of elements below for binary search

-15, -6, 0, 7, 9 , 23, 54, 82 and **Key=54**

Solution:

Index	1	2	3	4	5	6	7	8
Array a elements	**-15**	**-6**	**0**	**7**	**9**	**23**	**54**	**82**

- Compute mid =(1+8)/2=4

- Compare key x with mid element i.e 82 =a[mid] i.e a[4]=7 not equal to key 54.
- if (x<a[mid]) then no
- so (x>a[mid]) as 54>7, so call BinarySearch(a,mid+1, l, x);

Index	5	6	7	8
Array a	**9**	**23**	**54**	**82**

BinarySearch(a,5,8,7) so array is divided in to two parts and we compare only the right subarray.

Index	1	2	3	4
Array a elements	**-15**	**-6**	**0**	**7**

In right subarray again compute mid element mid=(5+8)/2=6

Since a[mid]=23 is lesser than 54 .again recursively call the Binarysearch for right subarray

BinarySearch(a,mid+1, l, x) i.e BinarySearch(a,7, 6, 8);

Index	5	6
Array a elements	**9**	**23**

Index	7	8
Array a elements	**54**	**82**

Compute mid=(7+8)/2=7

A[7]=54 matches with the key.so return sussess with index 7 saying key 54 is found at position 7 in the array.

2.4 Finding the Maximum and minimum

Problem statement: The problem is to find the maximum and minimum items in a set of n elements.

Algorithm 1: Straight method.

```
Algorithm StraightMaxMin($a, n, max, min$)
// Set $max$ to the maximum and $min$ to the minimum of $a[1:n]$.
{
    $max := min := a[1]$;
    for $i := 2$ to $n$ do
    {
        if ($a[i] > max$) then $max := a[i]$;
        if ($a[i] < min$) then $min := a[i]$;
    }
}
```

Note: This algorithm takes 2n-2 comparisons**.**

Divide and conquer

```
Algorithm MaxMin($i, j, max, min$)
// $a[1:n]$ is a global array. Parameters $i$ and $j$ are integers,
// $1 \leq i \leq j \leq n$. The effect is to set $max$ and $min$ to the
// largest and smallest values in $a[i:j]$, respectively.
{
    if $(i = j)$ then $max := min := a[i]$; // Small($P$)
    else if $(i = j - 1)$ then // Another case of Small($P$)
        {
            if $(a[i] < a[j])$ then
            {
                $max := a[j]; min := a[i]$;
            }
            else
            {
                $max := a[i]; min := a[j]$;
            }
        }
        else
        {   // If $P$ is not small, divide $P$ into subproblems.
            // Find where to split the set.
                $mid := \lfloor (i + j)/2 \rfloor$;
            // Solve the subproblems.
                MaxMin($i, mid, max, min$);
                MaxMin($mid + 1, j, max1, min1$);
            // Combine the solutions.
                if $(max < max1)$ then $max := max1$;
                if $(min > min1)$ then $min := min1$;
        }
}
```

Now what is the number of element comparisons needed for MaxMin? If $T(n)$ represents this number, then the resulting recurrence relation is

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

When n is a power of two, $n = 2^k$ for some positive integer k, then

$$
\begin{aligned}
T(n) &= 2\,\underline{T(n/2)} + 2 \\
&= 2[\,2\,T(n/4) + 2] + 2 \\
&= 4\,\underline{T(n/4)} + 4 + 2 \\
&= 4\,[2\,T(n/8) + 2] + 4 + 2 \\
&= 8\,T(n/8) + 8 + 4 + 2 \\
&= : \\
&= 2^k\,T(n/2^k)\ 2^k + 2^{k-1} + 2^{k-2} + \ldots 2
\end{aligned}
$$

As T(1) = 0 and T(2)=1 , consider k-1

$$
\begin{aligned}
&= 2^{k-1}\,T(n/2^{k-1})\ \ \underset{\longleftrightarrow}{[2^{k-1} + 2^{k-2} + 2^{k-3} + \ldots\ldots 2]} \\
&= 2^{k-1}\,T(n/2^{k-1})\quad [2^k - 2] \\
&= 2^k/2\ T(n/2^k * 2)\ \ [2^k - 2]
\end{aligned}
$$

We know that n= 2 ^{k}, substitute it

$$
\begin{aligned}
&= n/2\ T(\cancel{2}\,\cancel{n}/\cancel{n})\ [n-2] \\
&= n/2\ T(2)\ (n-2)
\end{aligned}
$$

As T(2)=1

$$
\begin{aligned}
&= n/2*(n-2) \\
&= 3n/2-2
\end{aligned}
$$

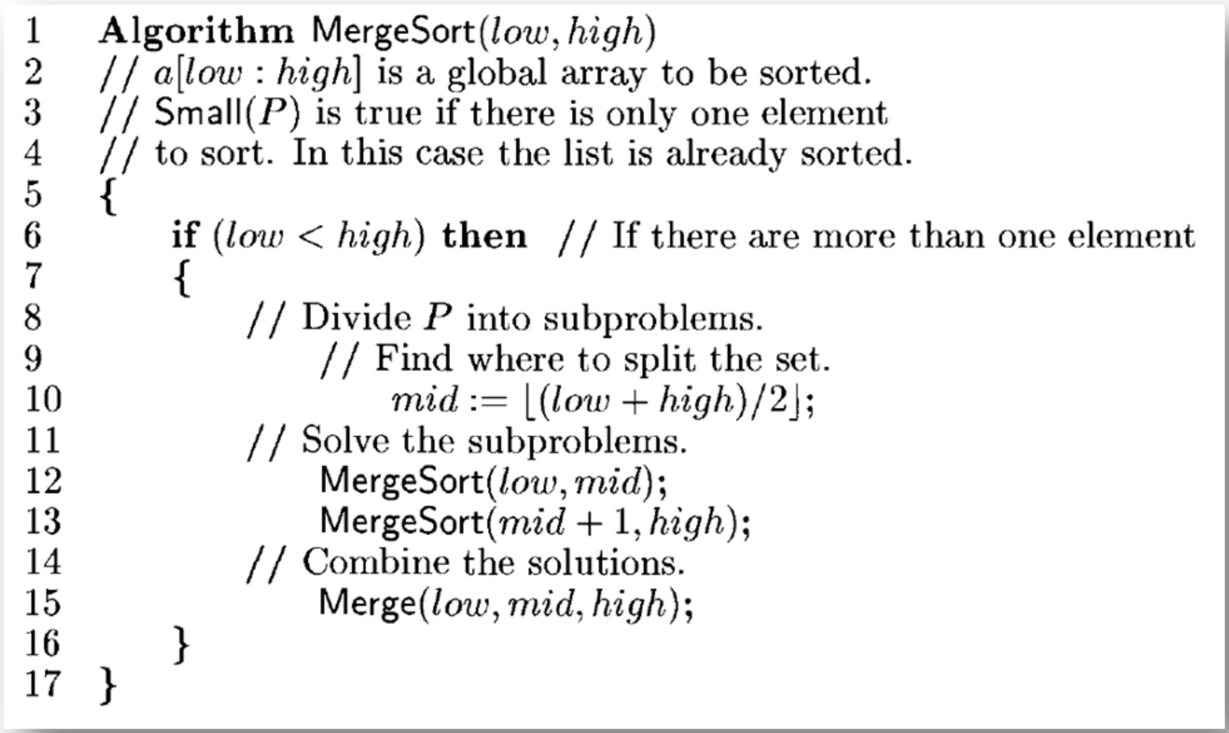
2.5 Merge Sort

Merge is a perfect example of successful application of divide and conquer strategy.

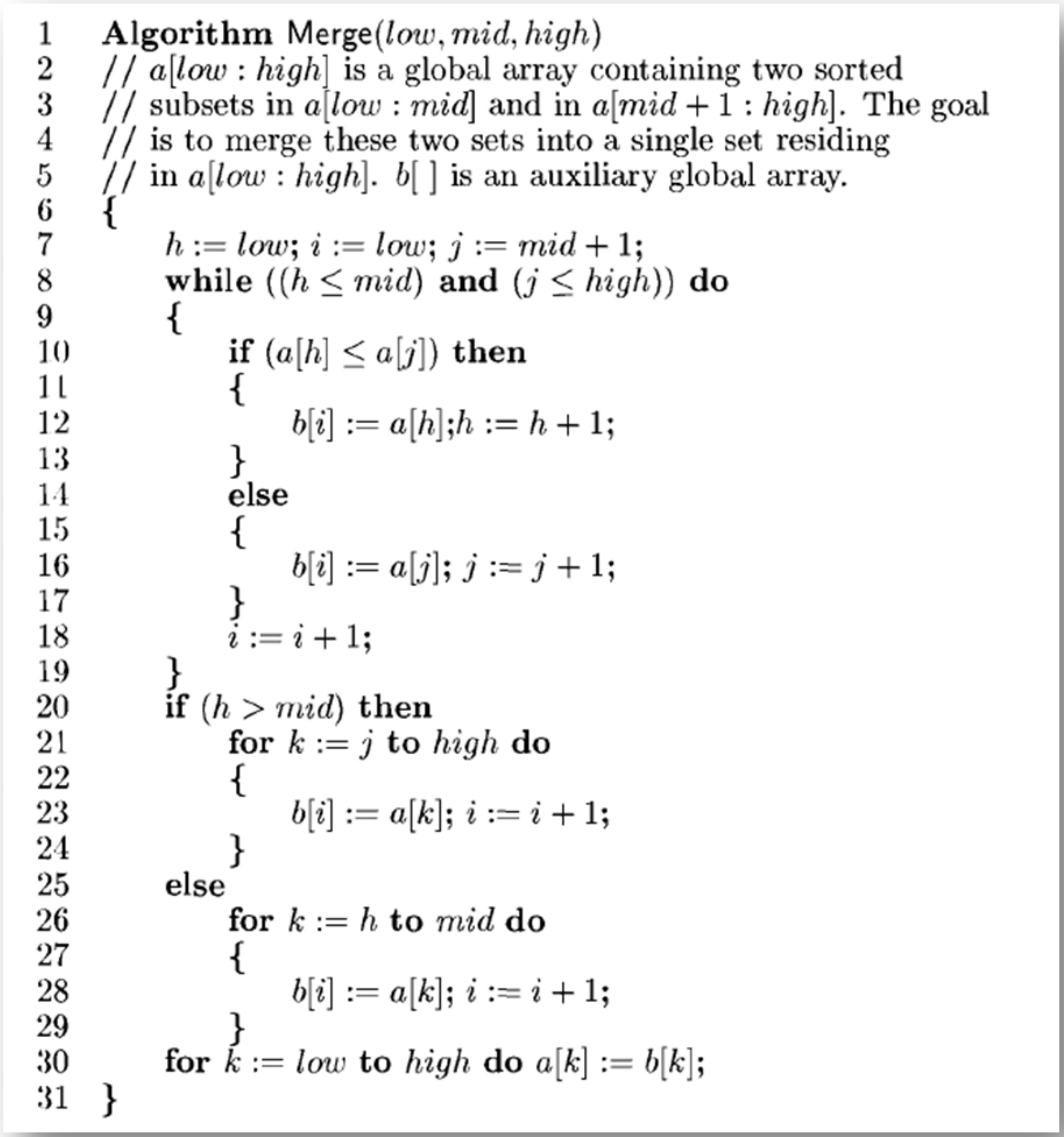
Given a sequence of n elements a[1],a[2],...,a[n], the merge sort algorithm will split into two sets a[1], ... a[⌊n/2⌋] and a[⌊n/2⌋+1],….,a[n]. Each set is individually sorted & resulting sorted sets are merged to get a single sorted array of n elements.

Procedure:

- **Divide**: Partition array in to two sub lists
- **Conquer**: Then sort two sub lists
- **Combine**: Merge sub problems



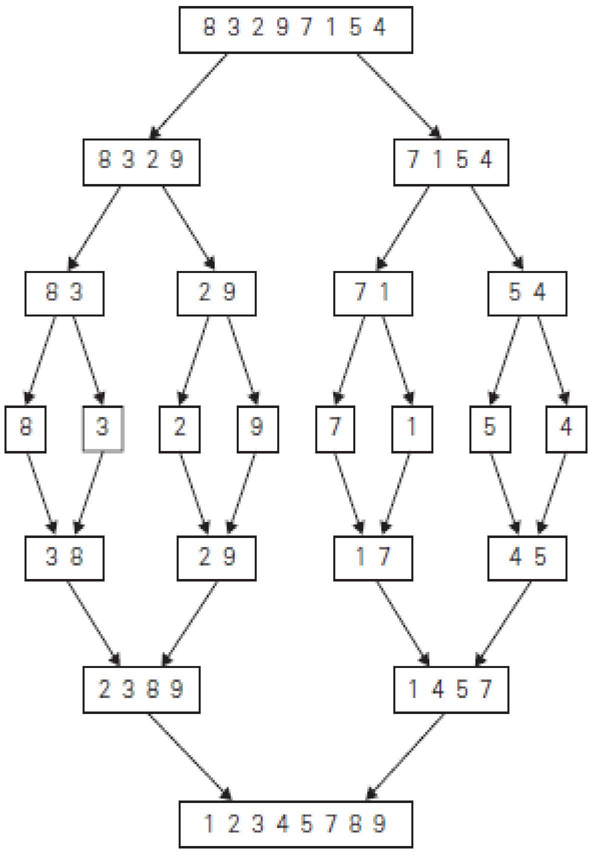
```
Algorithm MergeSort($low, high$)
// $a[low : high]$ is a global array to be sorted.
// Small($P$) is true if there is only one element
// to sort. In this case the list is already sorted.
{
    if ($low < high$) then  // If there are more than one element
    {
        // Divide $P$ into subproblems.
            // Find where to split the set.
                $mid := \lfloor (low + high)/2 \rfloor$;
        // Solve the subproblems.
            MergeSort($low, mid$);
            MergeSort($mid + 1, high$);
        // Combine the solutions.
            Merge($low, mid, high$);
    }
}
```



```
Algorithm Merge($low, mid, high$)
// $a[low : high]$ 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.
{
    $h := low$; $i := low$; $j := mid + 1$;
    while (($h \leq mid$) and ($j \leq high$)) do
    {
        if ($a[h] \leq a[j]$) then
        {
            $b[i] := a[h]$;$h := h + 1$;
        }
        else
        {
            $b[i] := a[j]$; $j := j + 1$;
        }
        $i := i + 1$;
    }
    if ($h > mid$) then
        for $k := j$ to $high$ do
        {
            $b[i] := a[k]$; $i := i + 1$;
        }
    else
        for $k := h$ to $mid$ do
        {
            $b[i] := a[k]$; $i := i + 1$;
        }
    for $k := low$ to $high$ do $a[k] := b[k]$;
}
```

Example: Consider array of elements **8 3 2 9 7 1 5 4**

The merge sort operation performed on it is depicted below.



Example of a merged operation

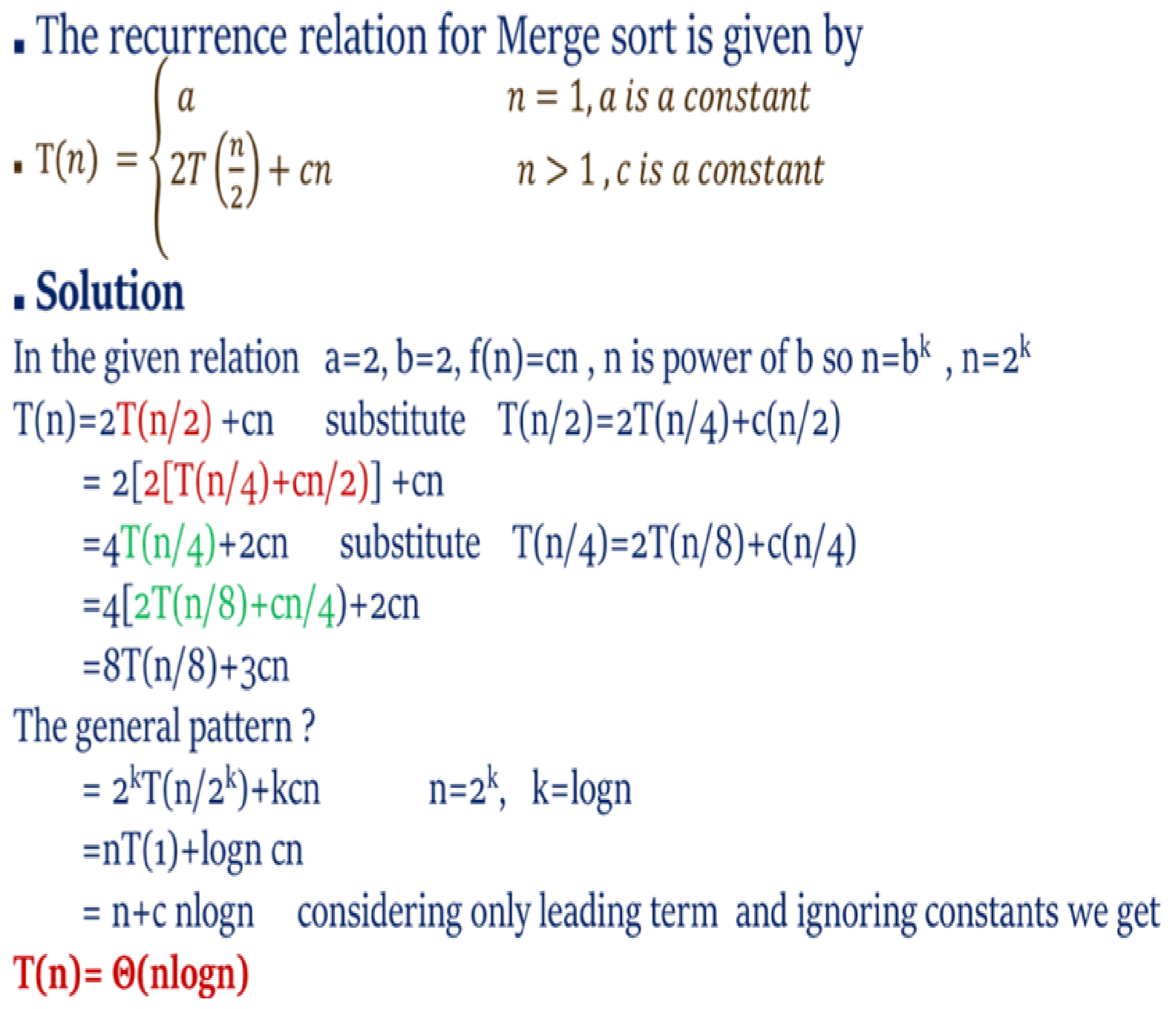
Analysis

Assuming for simplicity that *n* is a power of 2, the recurrence relation for the number of key comparisons *C (n)* is

$$C(n) = C(n/2) + C(n/2) + C_{merge}(n) \text{ for } n > 1,\ C(1) = 0.$$

To process **1st half** **2nd half** **combine**

$$C(n) = 2C(n/2) + C_{merge}(n) \text{ for } n > 1,\ C(1) = 0.$$



- The recurrence relation for Merge sort is given by
- $$T(n) = \begin{cases} a & n = 1, a \text{ is a constant} \\ 2T\left(\frac{n}{2}\right) + cn & n > 1, c \text{ is a constant} \end{cases}$$
- **Solution**

In the given relation a=2, b=2, f(n)=cn , n is power of b so $n=b^k$, $n=2^k$

$T(n)=2T(n/2)+cn$ substitute $T(n/2)=2T(n/4)+c(n/2)$

$= 2[2[T(n/4)+cn/2)]+cn$

$=4T(n/4)+2cn$ substitute $T(n/4)=2T(n/8)+c(n/4)$

$=4[2T(n/8)+cn/4)+2cn$

$=8T(n/8)+3cn$

The general pattern ?

$= 2^kT(n/2^k)+kcn$ $n=2^k$, $k=\log n$

$=nT(1)+\log n\ cn$

$= n+c\ n\log n$ considering only leading term and ignoring constants we get

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

2.6 Quick sort

Quick sort is the other important sorting algorithm that is based on the divide-and conquers approach. Unlike merge sort, which divides its input elements according to their position in the array, quick sort divides them according to their value.

A partition is an arrangement of the array's elements so that all the elements to the left of some element $A[s]$ are less than or equal to $A[s]$, and all the elements to the right of $A[s]$ are greater than or equal to it:

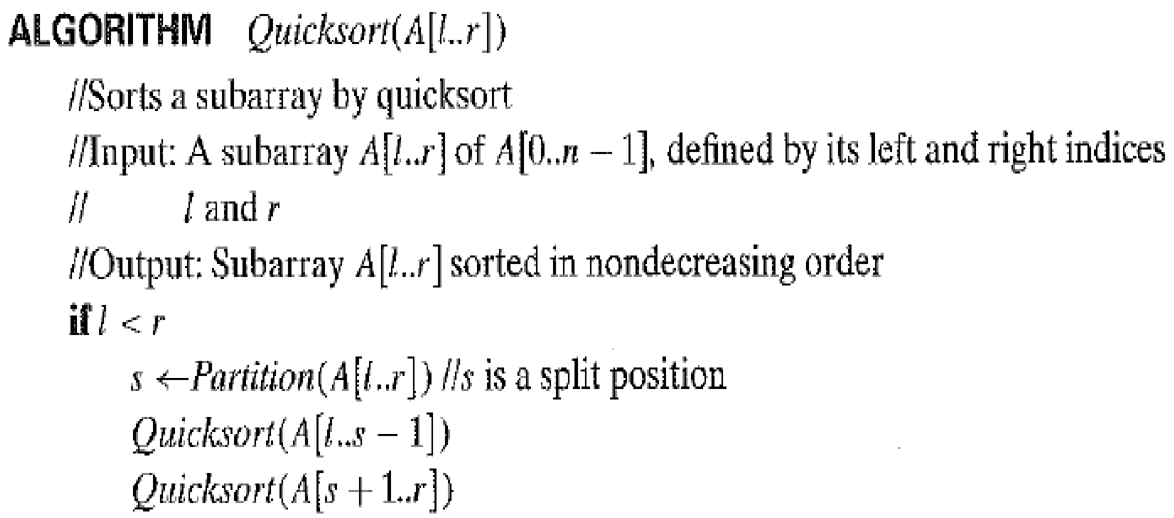
$$A[0] \ldots A[s-1] \text{ \_ \_\_ \_ all are } \le A[s]$$

$$A[s]\ A[s+1] \ldots A[n-1] \text{ \_ \_\_ \_all are } \ge A[s]$$

After a partition is achieved, *A*[*s*] will be in its final position in the sorted array, and we can continue sorting the two sub arrays to the left and to the right of *A*[*s*] independently (e.g., by the same method).

Note: the difference with merge sort: there, the division of the problem into two sub problems is immediate and the entire work happens in combining their solutions; here, the entire work happens in the division stage, with no work required to combine the solutions to the sub problems.

Here is pseudo code of quick sort:



```
ALGORITHM  Quicksort(A[l..r])
    //Sorts a subarray by quicksort
    //Input: A subarray A[l..r] of A[0..n − 1], defined by its left and right indices
    //        l and r
    //Output: Subarray A[l..r] sorted in nondecreasing order
    if l < r
        s ←Partition(A[l..r]) //s is a split position
        Quicksort(A[l..s − 1])
        Quicksort(A[s + 1..r])
```

Quick sort execution procedure:

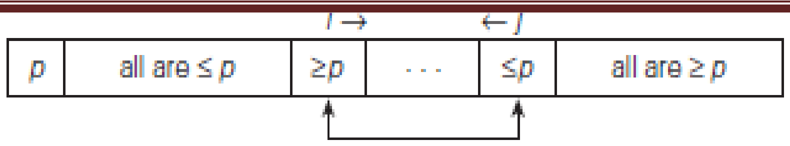
- Call to quick sort which performs sorting recursively for sub arrays.
- Partitioning or split position is identified by using partitioning function which divides the array in to two sub arrays.

Partitioning function procedure:

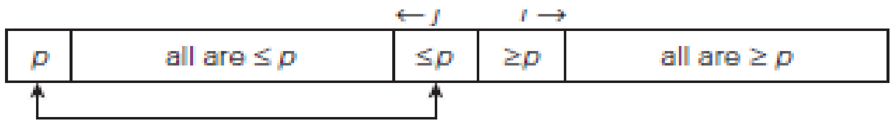
- 1. Make first element as the pivot element. **P** //many ways of selecting a pivot element exist.
- 2. set i (low index to point to 0)and j to high which is n-1.
- **Increment** i **until** $A[i] \geq p$.If condition is met stop incrementing i.
- **Decrement** j **until** $A[j] \leq p$. If condition is met stop decrementing j
- 5.Compare the positions of i and j (three cases may arise,<,>,=)

  a) If i<j perform swap(A[i], A[j]) and continue incrementing i and from the same positions.

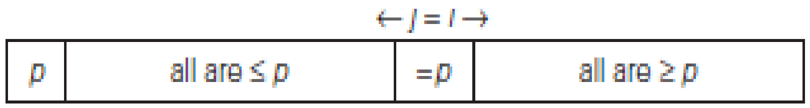
If scanning indices *i* and *j* have not crossed, i.e., $i < j$, we simply exchange *A*[*i*] and *A*[*j*] and resume the scans by incrementing *I* and decrementing *j*, respectively:



b) If i>j then swap pivot element with a[j].return j as the split position. If the scanning indices have crossed over, i.e., *i* > *j*, we will have partitioned the
subarray after exchanging the pivot with *A*[*j*]:



c) Finally, if the scanning indices stop while pointing to the same element, i.e., *i* = *j*,
the value they are pointing to must be equal to *p* (why?). Thus, we have the
subarray partitioned, with the split position *s* = *i* = *j* :

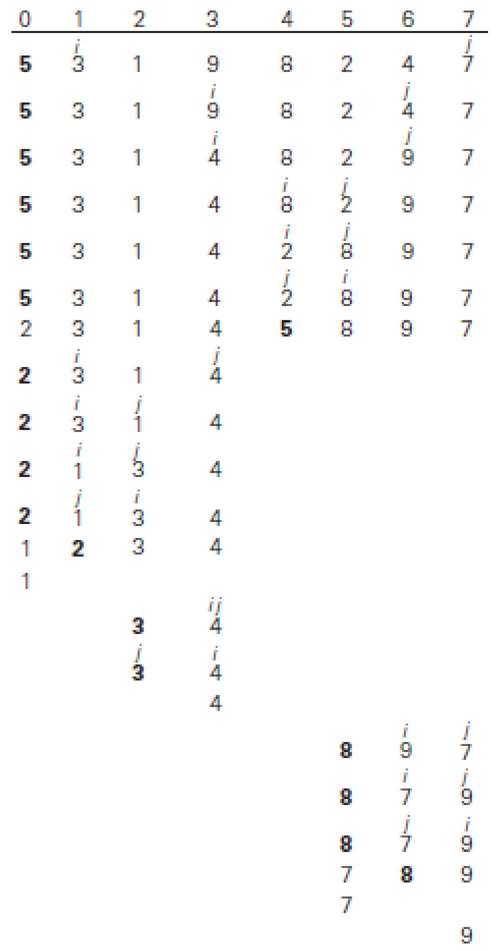


We can combine the last case with the case of crossed-over indices *(i > j)* by exchanging the pivot with *A*[*j*] whenever *i* ≥ *j* . j is the split position. Towards left of this positions elements will be lesser in value and right side elements will be larger in value. And the same are considered as two subarrays which we take separately to sort.

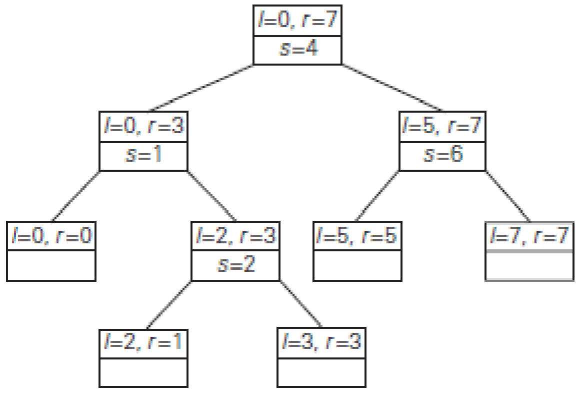
```
ALGORITHM   Partition(A[l..r])
    //Partitions a subarray by using its first element as a pivot
    //Input: A subarray A[l..r] of A[0..n − 1], defined by its left and right
    //        indices l and r (l < r)
    //Output: A partition of A[l..r], with the split position returned as
    //          this function's value
    p ← A[l]
    i ← l;   j ← r + 1
    repeat
        repeat i ← i + 1 until A[i] ≥ p
        repeat j ← j − 1 until A[j] ≤ p
        swap(A[i], A[j])
    until i ≥ j
    swap(A[i], A[j])   //undo last swap when i ≥ j
    swap(A[l], A[j])
    return j
```

Note that index *i* can go out of the subarray's bounds in this pseudocode. Rather than checking for this possibility every time index *i* is incremented, we can append to array*A*[0..*n* − 1]a "sentinel" that would prevent index *i* from advancing beyond position *n*. Note that the more sophisticated method of pivot selection mentioned at the end of the section makes such a sentinel unnecessary.

An example of sorting an array by quicksort is given in Figure.



(a)



(b)

Analysis:

Best case: If all the splits happen in the middle of corresponding sub arrays, we will have the best case. The number of key comparisons in the best case satisfies the recurrence

$$C_{best}(n) = 2C_{best}(n/2) + n \text{ for } n > 1,\ C_{best}(1) = 0.$$

From the recuurance relation,

$C(n)= 2\ C(n/2) + n$

$C(n)= 2[2\ C(n/4) + n] + n\ =\ 2^2\ C(n/2^2) + 2n$

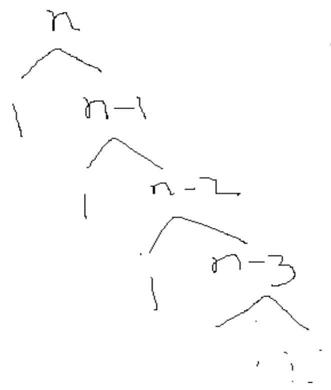
;

$C(n)=\ 2^k\ C(n/2^k) + kn$

We know $n = 2^k$, so $C(n) = n. C(n/n) + (\log_2 n)\, n = C(1) * n + n \log n = n\log n$

$$C_{best}(n) = n \log n.$$

Worst case: In the worst case, all the splits will be skewed to the extreme: one of the two subarrays will be empty, and the size of the other will be just 1 less than the size of the subarray being partitioned. So, after making $n + 1$ comparisons to get to this partition and exchanging the pivot $A[0]$ with itself, the algorithm will be left with the strictly increasing array $A[1..n - 1]$ to sort. The total number of key comparisons made will be equal to



$$C_{worst}(n) = 0 + n + T(n-1) = n(n+1)/2 = n^2$$

or

$$C_{worst}(n) = (n + 1) + n + . . . + 3 = (n + 1)(n + 2)/ 2 - 3 \in \_(n^2).$$

Average case: Let *Cavg(n)* be the average number of key comparisons made by quicksort on a randomly ordered array of size *n*. A partition can happen in any position s $(0 \le s \le n-1)$ after $n+1$comparisons are made to achieve the partition. After the partition, the left and right subarrays will have s and $n - 1 - s$ elements, respectively. Assuming that the partition split can happen in each position s with the same probability $1/n$, we get the following recurrence relation:

$$C_{avg}(n) = 1nn{-}1s{=}0[(n + 1) + C_{avg}(s) + C_{avg}(n - 1 - s)] \text{ for } n > 1,$$

$$C_{avg}(0) = 0, C_{avg}(1) = 0.$$

Its solution, which is much trickier than the worst- and best-case analyses, turns out to be

$$C_{avg}(n) \approx 2n \ln n \approx 1.39n \log2\, n.$$

Thus, on the average, quicksort makes only 39% more comparisons than in the best case. Moreover, its innermost loop is so efficient that it usually runs faster than mergesort on randomly ordered arrays of nontrivial sizes.

2.6 Stassen's matrix multiplication

Problem definition: Let A and B be two nxn matrices he product matrix C=AB is also an nxn matrix whose i, j th element is formed by taking the elements in the i th row of A and the j th column of B and multiplying them to get for all i and j between 1 to n.

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

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, is **$\theta(n^{3)}$.**

The divide and conquer strategy suggests another way to compute the product of two nxn matrices. For simplicity we assume that n is a power of two, that is, that there exists a non negative 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. If A and B are each partitioned in to four square sub matrices, each sub matrix having dimensions n/2 x n/2. Then the product AB can be computed by using the above formula for the product of two matrices. If and B are defined as below

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

then

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

To compute the AB using above formula we need to perform 8 multiplications of n/2 x n/2 matrices and four additions of n/2 x n/2 matrices. Since two n/2 x n/2 matrices can be added in C n^2 for the constant C. Therefore T(n)can be given as

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

Where b and c are constants.

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

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

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

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

where a and b are constants. Working with this formula, we get

$$\begin{aligned}
T(n) &= an^2[1+7/4+(7/4)^2+\cdots+(7/4)^{k-1}]+7^kT(1) \\
&\le cn^2(7/4)^{\log_2 n}+7^{\log_2 n},\ c \text{ a constant} \\
&= cn^{\log_2 4+\log_2 7-\log_2 4}+n^{\log_2 7} \\
&= O(n^{\log_2 7}) \approx O(n^{2.81})
\end{aligned}$$

Compared to regular multiplication, this method uses 7 multiplication and 18 additions or subtractions. In this method first we compute P,Q ,R,S T ,U and V (7 multiplications) and 10 matrix additions or subtractions

1. Consider the following matrices and compute the product using Strassen's Matrix multiplication:

$$A=\begin{bmatrix} 2 & 4 \\ 3 & 5 \end{bmatrix} \qquad B=\begin{bmatrix} 1 & 3 \\ 4 & 7 \end{bmatrix}$$

P= (2+5) (1+7)= 7* 8= 56

Q= (3+5)* 1= 8

R= 2* (3-7)= -8

S= 5*(4-1)= 15

T= (2+4) *7= 42

U= (3-2)* (1+3) = 4

V= (4-5)* (4+7)= -11

C11 = 56+ 15-42-11= 18

C12= -8+ 42= 34

C21= 8+15 = 23

$$C22 = 56-8-8+4 = 44$$

2.7 Advantages and disadvantages of divide and conquer

Divide and conquer method is a top down technique for designing an algorithm which consists of dividing the problem in to smaller sub problems hoping that the solutions of the sub problems are easier to find. The solution of all smaller problems is then combined to get a solution for the original problem.

Advantages:

- The difficult problem is broken down to sub problems and each problem is solved separately and independently. This is useful for obtaining solutions in easier way for difficult problems.
- This technique facilitates the discovers of new efficient algorithms. Example: Quick sort, Merge sort etc
- The sub problems can be executed on parallel processor.
- Hence time complexity can be reduced.

Disadvantages

- Large number of sub lists are created and need to be processed
- This algorithm makes use of recursive methods and the recursion is slow and complex.
- Difficulties in solving larger size of inputs