Module II

DIVIDE AND CONQUER

Contents

1.Write the general method for divide and conquer technique?

Divide and Conquer is one of the best-known general algorithm design technique It works according to the following general plan:

- 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.
- Now smaller and smaller subproblems of the same kind are generated until eventually subproblems that are small enoughto be solved without spiltting are produced.
- A typical case with k=2 is diagrammatically shown below

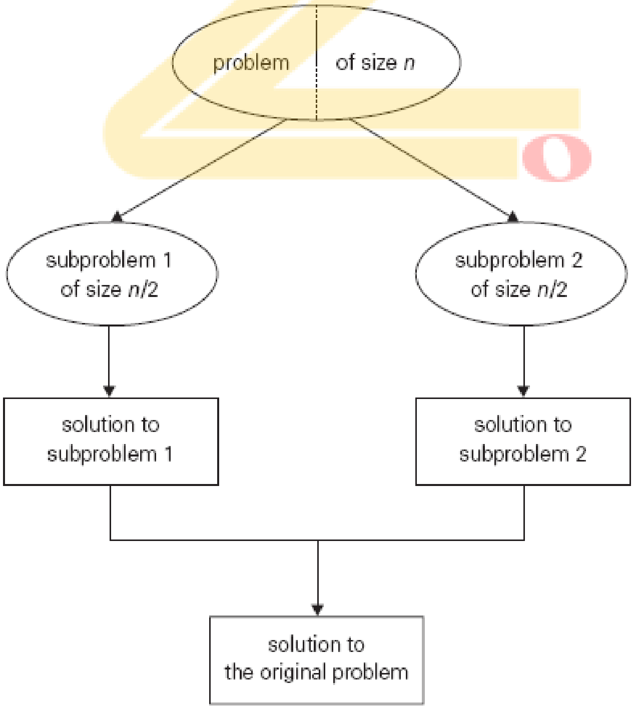


Figure: Divide-and-conquer technique (typical case).

2. Wrie the Control Abstraction of Divide and conquer?

- We can also write a control abstraction that illustrates the way on algorithm based on divide- and-conquer will look.
- A general divide and conquer design strategy(control abstraction) is illustrated as given below-

```
Algorithm DAndC (P)
{
    if small(P) then return S(P) //termination
    condition else
     {
          divide P into smaller instances $P_1$, $P_2$, $P_3$… $P_k$ k≥1; or 1≤k≤n
          Apply DAndC to each of these sub problems.
          return Combine (DAndC($P_1$), DAndC ($P_2$), DAndC ($P_3$)…DAndC ($P_k$));
     }
}
```

In the above specification,

- Initially ***DAndC(P)*** is invoked, where _P‘ is the problem 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. If small (P) is true then function **‘S’** is invoked.
- Otherwise the problem _P‘ is divided into sub problems. These sub problems are solved by recursive application of **DAndC**.
- Finally the solution from k sub problems is combined to obtain the solution of the given problem using **Combine** function.

Recurrence Equation for Divide and Conquer

- If the size of problem _p‘ is n and the sizes of the _k‘ sub problems are n_1, n_2 ….n_k, respectively, then the computing time of divide and conquer 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 divide and conquer method on any input of size n and
- ✓ g(n) is the time to compute answer directly for small inputs.
- ✓The function f(n) is the time for dividing the problem _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 describe them by using recursion.
- More generally, an instance of size **n** can be divided into **b** instances of size **n/b**, with **a** of them needing to be solved. (Here, a and b are constants; **a>=1 and b > 1**.).
- Assuming that size **n** is a power of **b** (i.e. n = b^k), to simplify our analysis, we get the following recurrence for the running time T(n):

Where f(n) is a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions.

Substitution Method - One of the methods for solving the recurrence relation is called the substitution method. This method repeatedly makes substitution for each occurrence of the function T in the right hand side until all such occurrences disappear.

Master Theorem - The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the master theorem.

It states that, in recurrence equation **T(n) = aT(n/b) + f (n)**, If f (n)∈Θ (n^d) where d ≥ 0 then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Analogous results hold for the *O* and Ω notations, too.

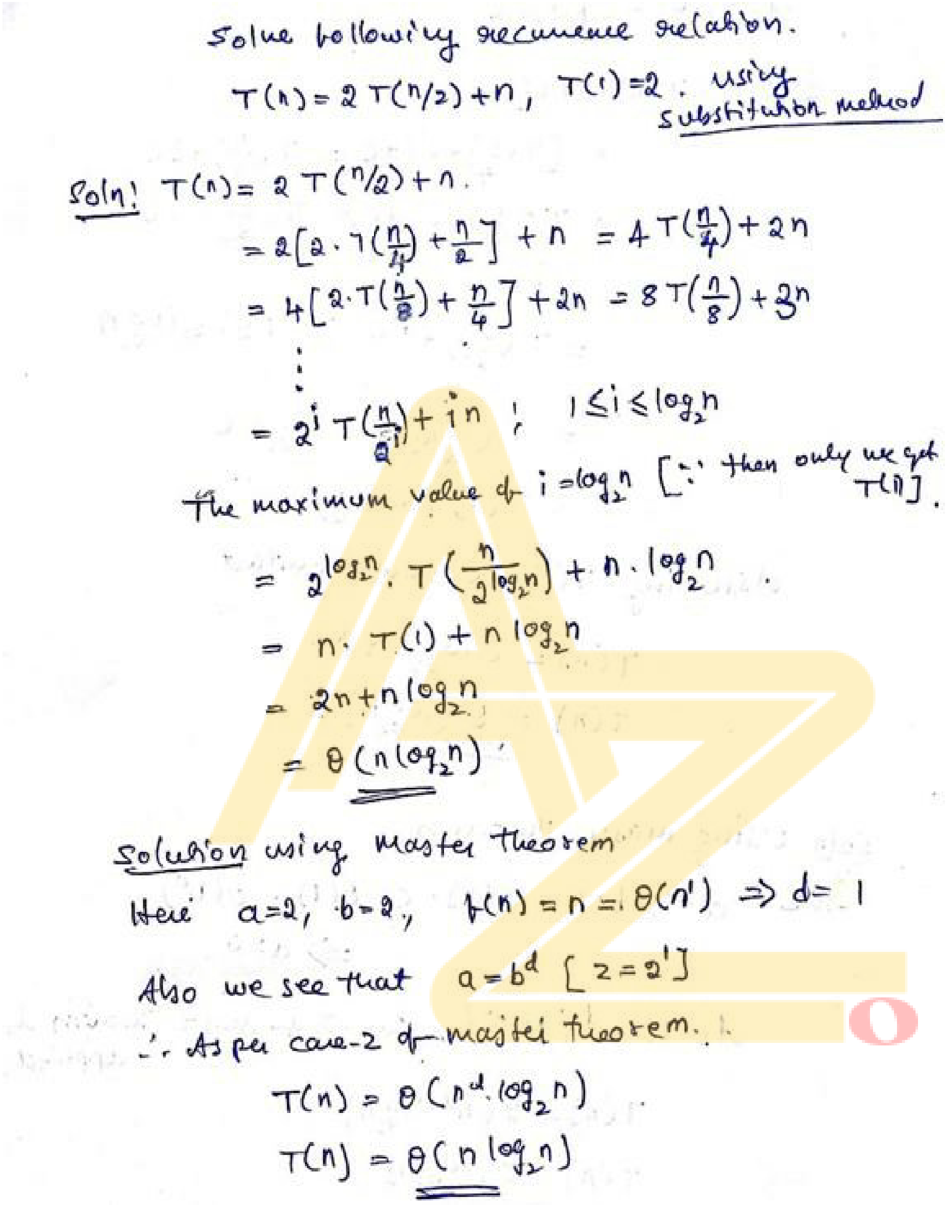
For example, the recurrence for the number of additions A(n) made by the divide-and conquer sum-computation algorithm on inputs of size n = 2^k is

Thus, for this example, a = 2, b = 2, and d = 0; hence, since a >b^d,

$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

3.Solve master theorm ?

Problems on Substitution method & Master theorem to solve the recurrence relation



Solve following recurrence relation.

$T(n) = 2T(n/2) + n$, $T(1) = 2$. using substitution method

Soln: $T(n) = 2T(n/2) + n$.

$= 2[2\cdot T(\frac{n}{4}) + \frac{n}{2}] + n = 4T(\frac{n}{4}) + 2n$

$= 4[2\cdot T(\frac{n}{8}) + \frac{n}{4}] + 2n = 8T(\frac{n}{8}) + 3n$

$\vdots$

$= 2^i T(\frac{n}{2^i}) + in$; $1 \le i \le \log_2 n$

The maximum value of $i = \log_2 n$ [∵ then only we get $T(1)$].

$= 2^{\log_2 n} \cdot T(\frac{n}{2^{\log_2 n}}) + n \cdot \log_2 n$.

$= n \cdot T(1) + n \log_2 n$

$= 2n + n \log_2 n$

$= \theta(n \log_2 n)$

Solution using master theorem

Here $a=2$, $b=2$, $f(n) = n = \theta(n^1) \Rightarrow d = 1$

Also we see that $a = b^d$ $[2 = 2^1]$

∴ As per case-2 of master theorem.

$T(n) = \theta(n^d . \log_2 n)$

$T(n) = \theta(n \log_2 n)$

$$T(n) = T(\tfrac{n}{2}) + n$$
$$= T(\tfrac{n}{4}) + \tfrac{n}{2} + n$$
$$= T(\tfrac{n}{8}) + \tfrac{n}{4} + \tfrac{n}{2} + n$$
$$\vdots$$
$$= T\left(\frac{n}{2^i}\right) + \left(\frac{n}{2^{i-1}} + \frac{n}{2^{i-2}} + \cdots \frac{n}{4} + \frac{n}{2} + n\right)$$
$$1 \leq i \leq \log_2 n$$
$$= T(1) + \left(\frac{n}{2^{\log_2 n}-1} + \frac{n}{2^{\log_2 n - 2}} + \cdots \frac{n}{4} + \frac{n}{2} + n\right)$$
$$= T(1) + \frac{n}{(2^{\log_2 n}/2)} + \frac{n}{\left(\frac{2^{\log_2 n}}{2^2}\right)} + \cdots \frac{n}{4} + \frac{n}{2} + n$$

Assume $T(1) = 1$,

$$= 1 + 2 + 2^2 + \cdots 2^{\log_2 n - 2} + 2^{\log_2 n - 1} + 2^{\log_2 n}$$
$$= 2^{\log_2 n + 1} - 1 \qquad (\because 1 + 2 + 2^2 \cdots 2^k = 2^{k+1} - 1)$$
$$= 2^{\log_2 n} \cdot 2 - 1$$
$$= n \cdot 2 - 1 = \underline{\underline{2n-1}} \in \underline{\underline{\theta(n)}}$$

Soln using master theorem

Here $a = 1$, $b = 2$ $f(n) = n = \theta(n^1) \Rightarrow d = 1$.

Since $a < b^d$ $(1 < 2^1)$

$$T(n) = \theta(n^d)$$
$$= \underline{\underline{\theta(n)}}$$

⊛ Solve $T(n) = 2\cdot T\left(\frac{n}{2}\right)+1$

<u>Soln</u> $T(n) = 2\cdot\left[2\cdot T\left(\frac{n}{4}\right)+1\right]+1 = 4\,T\left(\frac{n}{4}\right)+(2+1)$

$$= 4\left[2\cdot T\left(\tfrac{n}{8}\right)+1\right]+(2+1)$$

$$= 8\cdot T\left(\tfrac{n}{8}\right)+(4+2+1)$$

$$\vdots$$

$$= 2^i T\left(\tfrac{n}{2^i}\right)+(2^{i-1}+2^{i-2}+\dots 2+1)$$

$$1 \le i \le \log_2 n.$$

$$= 2^{\log_2 n}\cdot T(1)+(2^i-1)$$

$$= n\cdot T(1)+2^{\log_2 n}-1$$

Assuming $T(1)=1$

$$= n+n-1$$

$$= \underline{\underline{2n-1}}$$

$$= \underline{\underline{\Theta(n)}}$$

<u>Soln</u> Using master theorem

$a=2,\ b=2,\ f(n)=1$
$= \Theta(1) = \Theta(n^0) \Rightarrow d=0.$

Since $a > b^d$ $(2 > 2^0)$, case-3 is applied

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

$$= \Theta(n^{\log_2 2})$$

$$= \underline{\underline{\Theta(n)}}$$

4.Explain Binary Search algorithm with explaination?

Problem definition: Let a_i, $1 \leq i \leq n$ be a list of elements that are sorted in non-decreasing order. The problem is to find whether a given element x is present in the list or not. If x is present we have to determine a value j (element‘s position) such that a_j=x. If x is not in the list, then j is set to zero.

Solution: For the above problem, the algorithm can be implemented as recursive or non- recursive algorithm.

✓ **Algorithm BinSearch (A, low , high, x)**

```
 //Given an array A[low : high] of elements in non decreasing order , 1≤ low ≤ high
//Determine whether x is present, and if so return j such that x=A[j]; else return 0.
  {
            if (low=high) then
            {
              if(x=A[high]) then return h; //If Small
              (P) else return 0;
            }
            else
            {//reduce P into a smaller sub
              problem. mid :=[(low +high)/2];
              if(x=A[mid]) then return
              mid; else if(x<A[mid]) then
                    return BinSearch (A, low, mid-1, x);
              else return BinSearch (A, mid+1, high,
              x);
            }
  }
```

The above algorithm is a **recursive binary search** algorithm.

✓ **Explanation:**

The problem is subdivided into smaller problems until only one single element is left out.

1. If low = high then it means there is only one single element. So compare it with the search element ‗x‘. if both are equal then return the index, else return 0.
2. If low ≠ high then divide the problem into smaller subproblems.
   a. Calculate mid = (low+high)/2.
   b. Compare x with element at mid if both are equal the return index mid.

c. Else if x is less than A[mid] then the element x would be in the first partition A[low:mid-1]. So make a recursive call to BinSearch with low as low and high as mid-1.

d. Else x is greater than A[mid] then the element x would be in the second partition A[mid+1: high]. So make a recursive call to BinSearch with low as mid+1 and high as high.

✓ The below algorithm is an iterative algorithm for Binary search.

Algorithm BinSearch (a, n, x)

```
//Given an array a[1: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
{
    low:=1;
    high:=n;
    while(low ≤ high) do
    {
            mid:=[(low+high)/2];
            if(x<a[mid])then high:=mid-1;
            else if(x>a[mid])then
            low:=mid+1; else return mid;
    }
    return 0;
}
```

Example:-

Consider the elements :

-15,-6,0,7,9,23,25,54,82,101,112,125,131,142,151.

→ Place them in a[1:14], and simulate the steps BinSearch goes through as it searches for different values of ‗x'.

→ Only the variables, low, high & mid need to be traced as we simulate the algorithm.

We try the following values for x: 151, -14 and 9. For 2 successful searches & 1 for unsuccessful search.

- Table shows the traces of Bin search on these 3 steps.

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

Theorem: Algorithm Binsearch(a,n,x) works correctly.

Proof: We assume that all statements work as expected and that comparisons such as x>a[mid] are appropriately carried out.

- Initially low =1, high= n, n>=0, and a[1]<=a[2]<=……..<=a[n].
- If n=0, the while loop is not entered and is returned.
- Otherwise we observe that each time through the loop the possible elements to be checked of or equality with x are a[low], a[low+1],……..,a[mid],……a[high].
- If x=a[mid], then the algorithm terminates successfully.
- Otherwise, the range is narrowed by either increasing low to (mid+1) or decreasing high to (mid-1).
- Clearly, this narrowing of the range does not affect the outcome of the

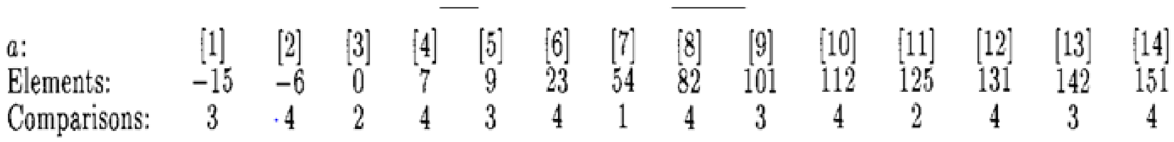
search. • If low becomes > than high, then ‗x' is not present & hence the loop is exited.

Theorem: Algorithm Binsearch(a,n,x) works correctly.

Proof: We assume that all statements work as expected and that comparisons such as x>a[mid] are appropriately carried out.

- Initially low =1, high= n, n>=0, and a[1]<=a[2]<=……..<=a[n].
- If n=0, the while loop is not entered and is returned.

- Otherwise we observe that each time through the loop the possible elements to be checked of or equality with x are a[low], a[low+1],……..,a[mid],……a[high].
- If x=a[mid], then the algorithm terminates successfully.
- Otherwise, the range is narrowed by either increasing low to (mid+1) or decreasing high to (mid-1).
- Clearly, this narrowing of the range does not affect the outcome of the
- search. If low becomes > than high, then ‗x' is not present & hence the loop is exited.



a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
Elements:	−15	−6	0	7	9	23	54	82	101	112	125	131	142	151
Comparisons:	3	4	2	4	3	4	1	4	3	4	2	4	3	4

- Each and every internal node represents a successful search and the values denoted by internal nodes are the various values taken by the variable mid.
- If the value of mid is at level zero then the no. of comparisons for an element present at the position is one.
- The no. of comparisons for a successful search if the mid value is at level one is two and the maximum number of comparisons for a successful search in the above decision tree is equal to four. i.e., the max. No. of comparisons is directly proportional to the height of the tree. Therefore the time complexity for a successful search is given by **O (log$_2$ n).**
- Each and every unsuccessful search terminates at an external node. The no. of comparisons needed for an unsuccessful search of an element less than -15 is 3. For the entire remaining unsuccessful search the no. of comparisons made is 4. Therefore the average no. of elemental comparisons of an unsuccessful search is

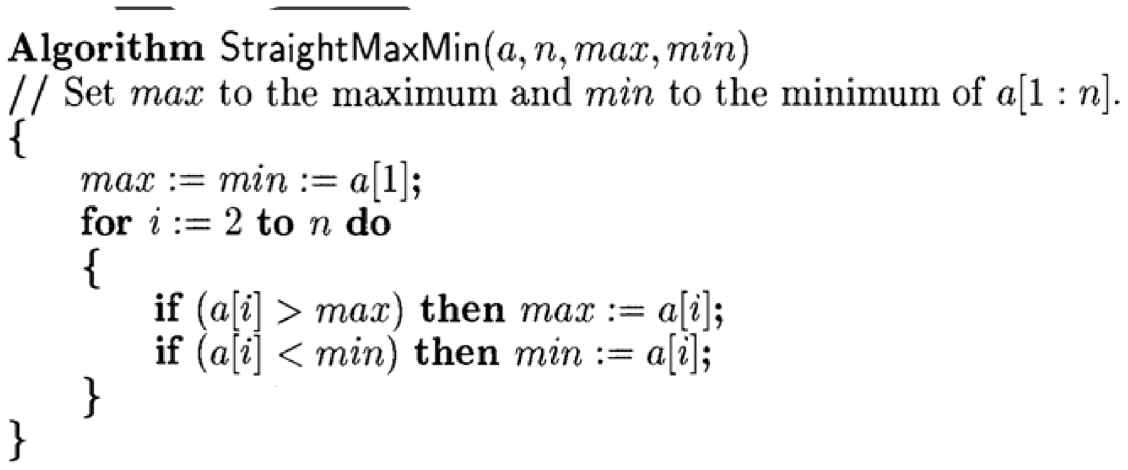
$$\begin{bmatrix} 3*1+4*14 \\ 15 \end{bmatrix} = \begin{bmatrix} 59 \\ 15 \end{bmatrix} = 3.93 \cong 4$$

- Except for one case the no. of comparisons for unsuccessful search is constant and is equal to the height of the tree. Therefore the time for unsuccessful search is given by **O (log$_2$ n).**

5.Write an algorithm for Finding The Maximum And Minimum

Problem statement: Given a list of n elements, the problem is to find the maximum and minimum items.

StraightMaxMin: A simple and straight forward algorithm to achieve this is given below.



```
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 $min := a[i]$;
    }
}
```

Algorithm for maximum and minimum

Explanation:

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

  if(a[i]>max) then max =
  a[i]; else if (a[i]< min)
  min=a[i];

- Now the best case occurs when the elements are in increasing order, then only first if condition is satisfied(i,e true) therefore the number of comparisions is (n -1).
- The worst case condition occurs when the elements are in decreasing order, in this case the first condition is evaluated as false and the second condition is evaluated to true therefore the number of comparisions is 2(n-1).

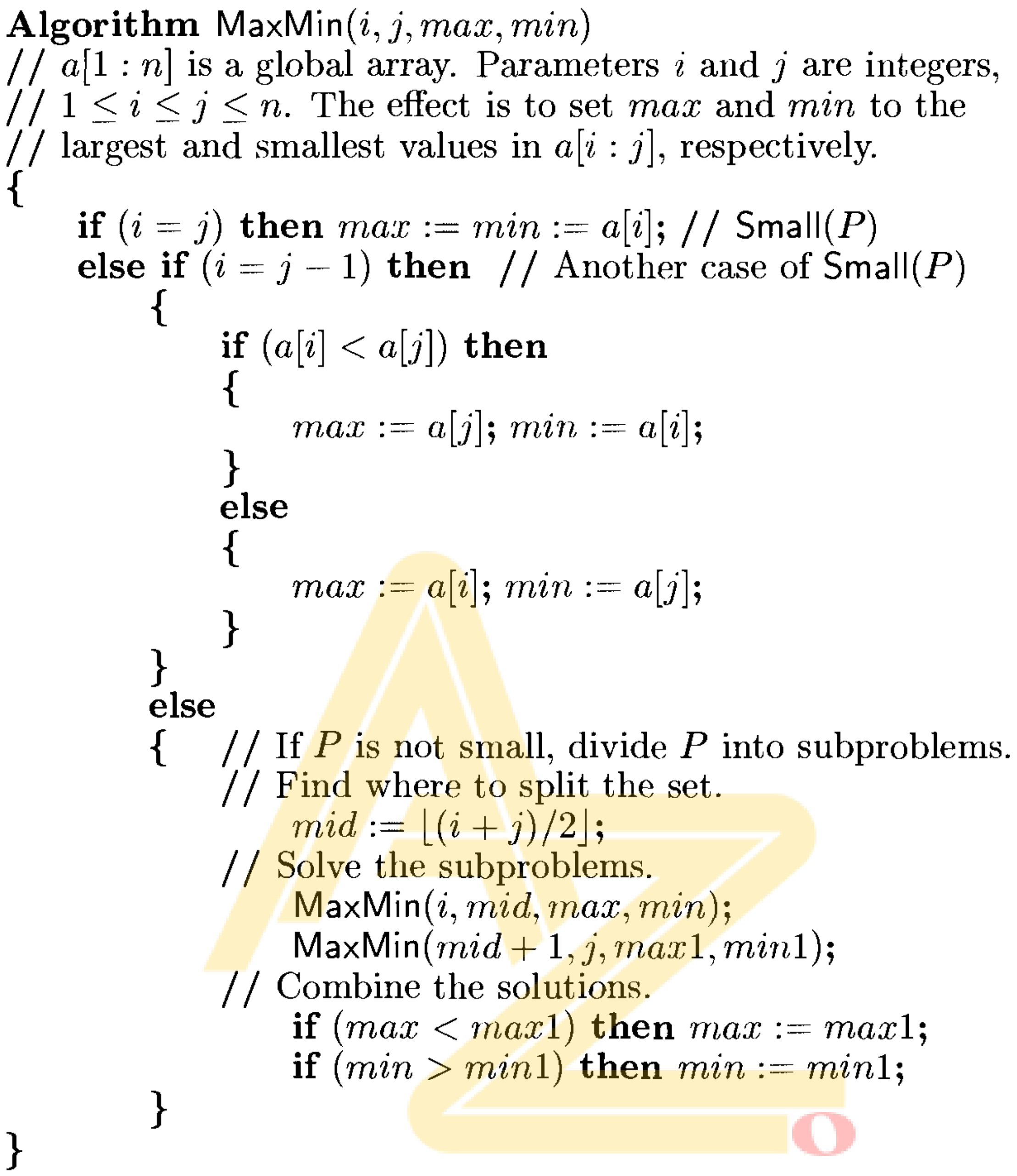
Algorithm based on Divide and Conquer strategy

- Let P = (n, a [i],……,a[j]) denote an arbitrary instance of the problem. Here ‗n' is the no. of elements in the list (a[i],….,a[j]) and we are interested in finding the maximum and minimum of the list.

- Let Small(P) be true when <=2. In this case, the maximum and minimum are a[i] if n=1 and if n=2, the problem can be solved by making one comparison.

- If the list has more than 2 elements, P has to be divided into smaller instances. For example, we might divide ‗P‘ into the 2 instances, P1=([n/2],a[1],……..a[n/2]) and P2= (n-[n/2], a[[n/2]+1],….., a[n])
- After having divided ‗P‘ into 2 smaller sub problems, we can solve them by recursively invoking the same divide-and-conquer algorithm.
- When the maximum and minimum of these sub problems are determined, the two maxima are compared and the two minima are compared to achieve the solution for the entire problem.

The algorithm is as follows:



```
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 $min := min1$;
        }
}
```

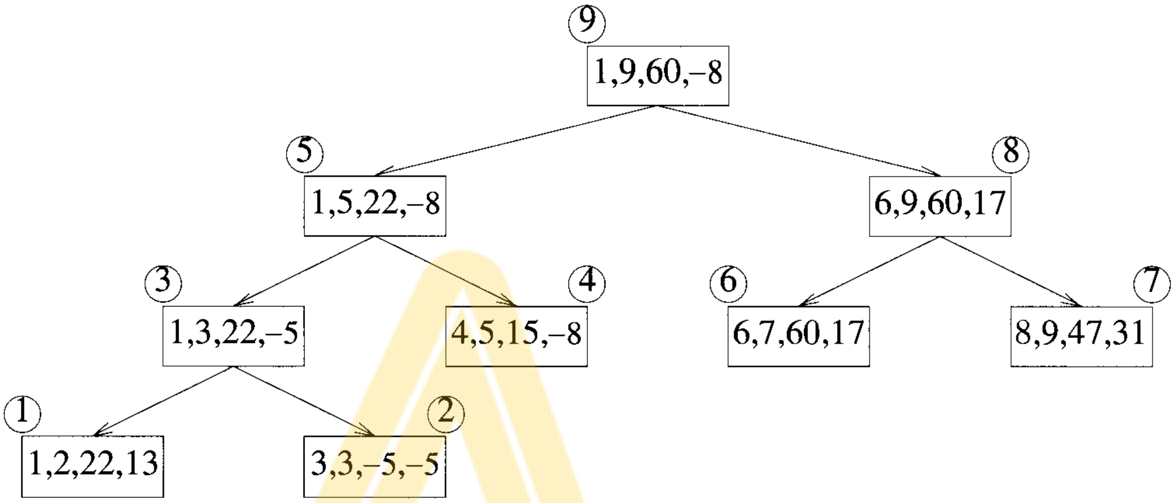
- The procedure is initially invoked by the statement MaxMin(1,n,x,y). for this node has four items of information: i, j, max, min.
- Suppose we simulate MaxMin on the following nine elements:

a: [1] [2] [3] [4] [5] [6] [7] [8] [9]

22 13 -5 -8 15 60 17 31 47

- A good way of keeping track of recursive calls is to build a tree by adding a node each time a new call is made. On the array a[ ] above, the following tree is produced.

- We see that the root node contains 1 and 9 as the values of i and j corresponding to the initial call to MaxMin. This execution produces two new call to MaxMin, where i and j have the values 1, 5 and 6, 9, and thus split the set into two subsets of the same size. From the tree we can immediately see that the maximum depth of recursion is four (including the first call). The circled numbers in the upper left order of each node represent the the orders in which max and min are assigned values.



Time Complexity:

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

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

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

.

.

.

= 2^{k-1} T(2) + ∑(1≤i≤k-1) 2^k

= $2^{k-1 +}$ 2^k – 2

= 3n/2 – 2 = O(n)

- Note that 3n/2 – 2 is the best, average, worst case number of comparison when n is a power of two.

Comparisons with Straight Forward Method:

Compared with the 2n – 2 comparisons for the Straight Forward method, this is a saving of 25% in comparisons. It can be shown that no algorithm based on comparisons uses less than 3n/2 – 2 comparisons.

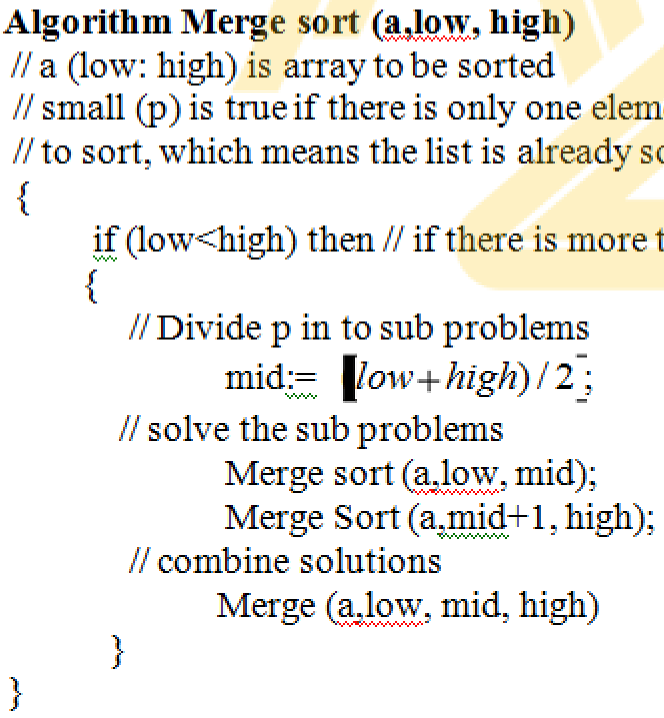
Space Complexity

6.Write Merge Sort algorithm? And explain in deatil

- Merge sort is a perfect example of a successful application of the divide-and conquer technique.

- It sorts a given array A[1],….,A[n] by dividing it into two halves A[1],…..,A[_n/2_] and A[_n/2_ +1],….,A[n], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.
- The following algorithm describes the process using recursion and the algorithm Merge

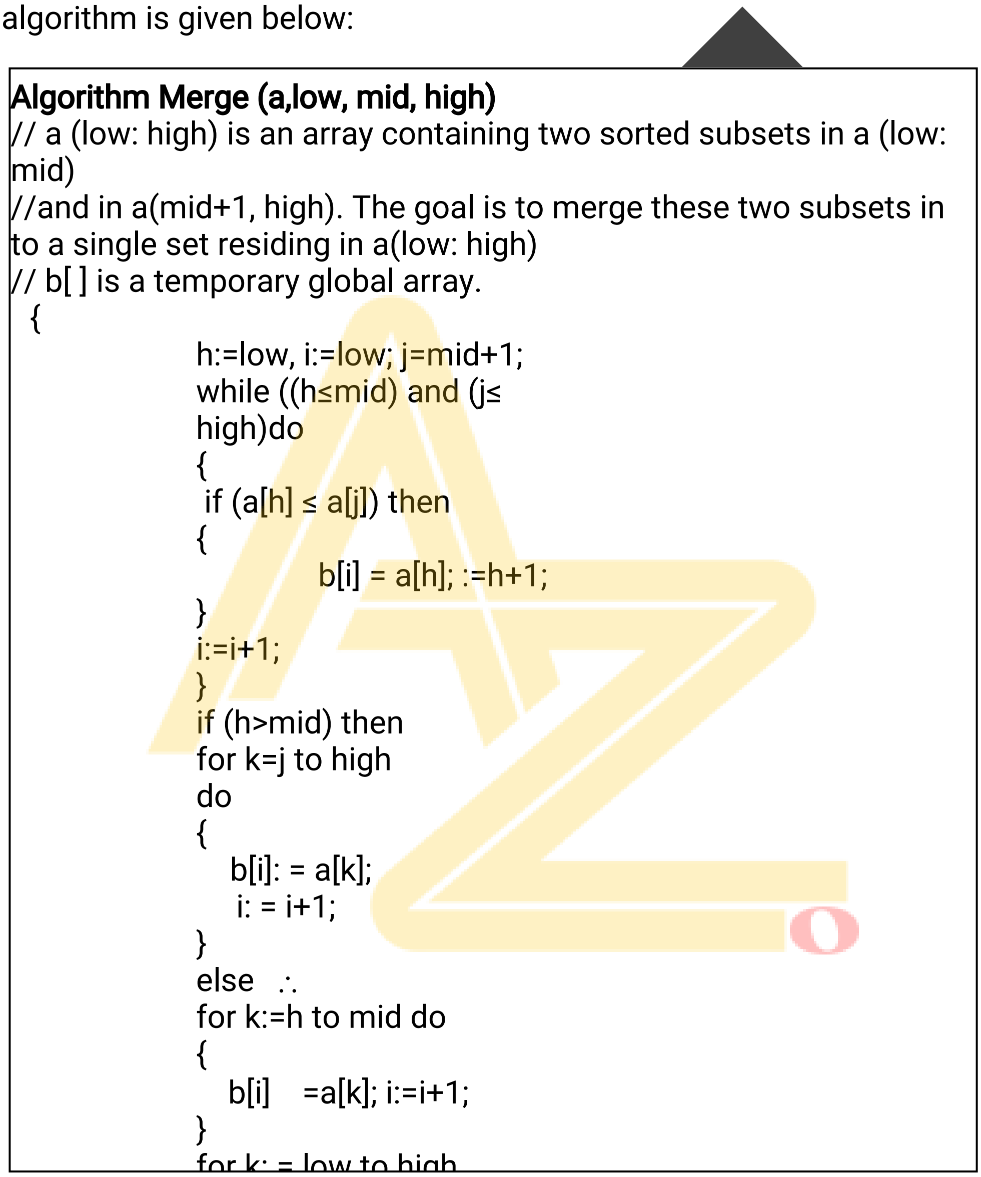
merges two sorted sets.



```
Algorithm Merge sort (a,low, high)
// a (low: high) is array to be sorted
// small (p) is true if there is only one element
// to sort, which means the list is already sorted
{
       if (low<high) then // if there is more than one element
      {
          // Divide p in to sub problems
                   mid:= $\lfloor (low+high)/2 \rfloor$;
         // solve the sub problems
                   Merge sort (a,low, mid);
                   Merge Sort (a,mid+1, high);
          // combine solutions
                  Merge (a,low, mid, high)
         }
}
```

- The *merging* of two sorted arrays can be done as follows. Two pointers (array indices) are initialized to point to the first elements of the arrays being merged.

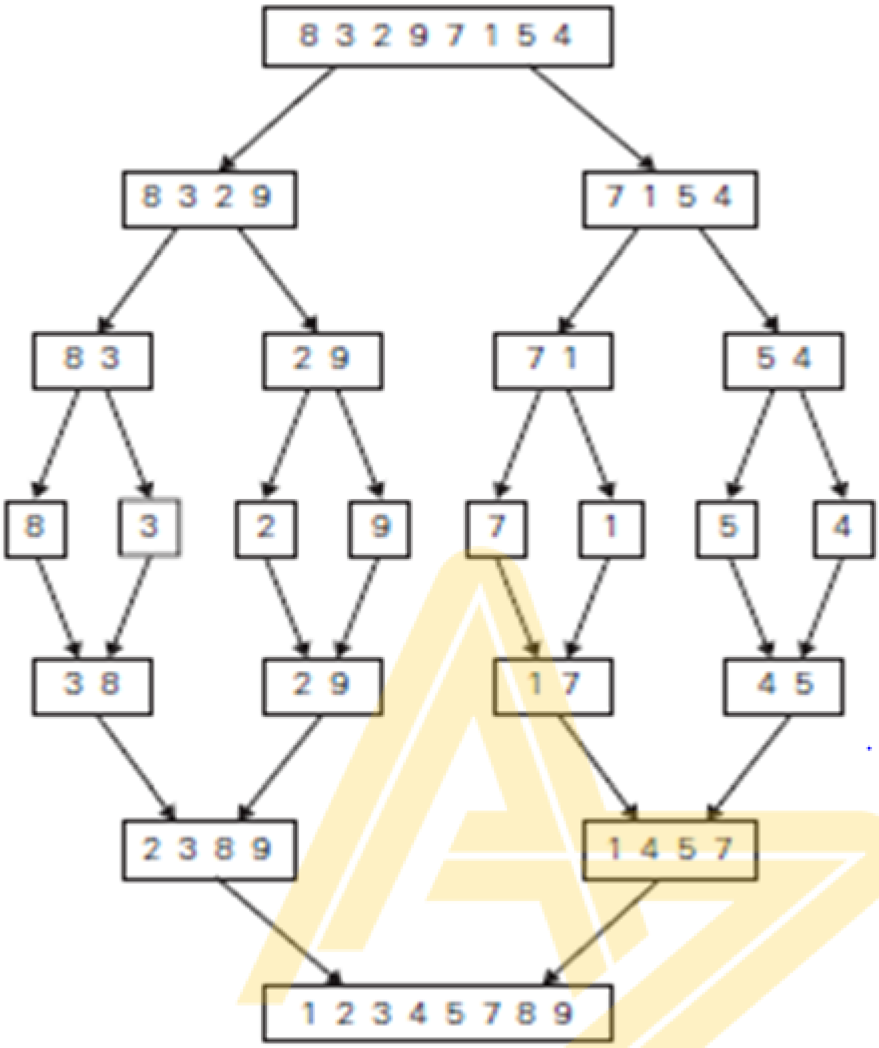
- The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from.
- This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.
- The algorithm is given below:



Algorithm Merge (a,low, mid, high)

```
// a (low: high) is an array containing two sorted subsets in a (low:
mid)
//and in a(mid+1, high). The goal is to merge these two subsets in
to a single set residing in a(low: high)
// b[ ] is a temporary global array.
  {
                 h:=low, i:=low; j=mid+1;
                 while ((h≤mid) and (j≤
                 high)do
                 {
                  if (a[h] ≤ a[j]) then
                 {
                             b[i] = a[h]; :=h+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
```

The operation of the algorithm on the list 8,3,2,9,7,1,5,4 is illustrated below:



Analysis

- Here the basic operation is key comparison. As merge sort execution does not depend on the order of the data, best case and average case runtime are the same as worst case runtime.
- Assuming for simplicity that total number of elements **n** is a power of 2, the recurrence relation for the number of key comparisons *C*(*n*) is

  Where, $C_{merge}(n)$ is the number of key comparison made during the merging stage.
- Let us analyze $C_{merge}(n)$, the number of key comparisons performed during the merging stage. At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needing to be processed is reduced by 1. In the worst case, neither of the two arrays becomes empty before the other one contains just one element (e.g., smaller elements may come from the alternating arrays). Therefore, for the worst case, $C_{merge}(n) = n - 1$. Now,

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

Solving the recurrence equation using **master theorem:** Here a = 2, b = 2, f (n) = n, d = 1. Therefore $2 = 2^1$, case 2 holds in the master theorem

C_{worst} (n) = Θ (n^d log n) = Θ (n^1 log n) = Θ (n log n) Therefore **C_{worst}(n) = Θ (n log n)**

Advantages

- For large *n,* the number of comparisons made by this algorithm in the average case turns out to be about 0.25*n* less and hence is also in **Θ** *(n* log *n).*
- Mergesort is a stable algorithm

Limitations

- The principal shortcoming of mergesort is the linear amount [O(n)] of extra storage the algorithm requires. Though merging can be done in-place, the resulting algorithm is quite complicated and of theoretical interest only.

Variations of merge sort

1. The algorithm can be implemented bottom up by merging pairs of the array‘s elements, then merging the sorted pairs, and so on. (If n is not a power of 2, only slight bookkeeping complications arise.) This avoids the time and space overhead

of using a stack to handle recursive calls.

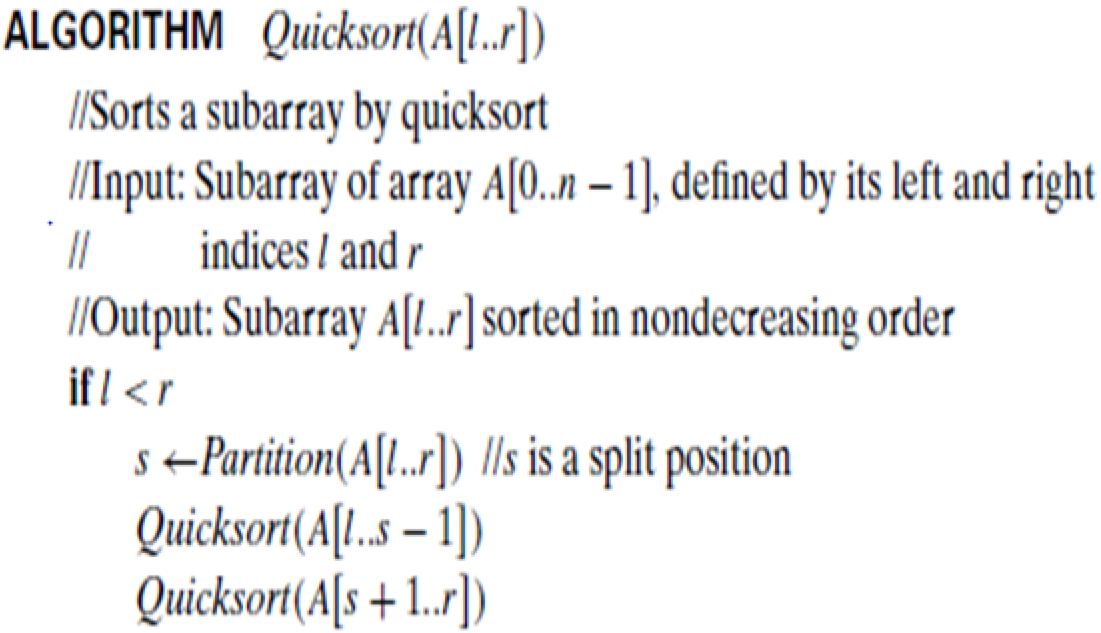
2. We can divide a list to be sorted in more than two parts, sort each recursively, and then merge them together. This scheme, which is particularly useful for sorting files residing on secondary memory devices, is called multiway mergesort.

Quick Sort

- Quicksort is the other important sorting algorithm that is based on the divide-andconquer approach. Unlike mergesort, which divides its input elements according to their position in the array, quicksort 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

$$\underbrace{A[0]\dots A[s-1]}_{\text{all are } \le A[s]}\quad A[s]\quad \underbrace{A[s+1]\dots A[n-1]}_{\text{all are } \ge A[s]}$$

- Obviously, after a partition is achieved, A[s] will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and to the right of A[s] independently (e.g., by the same method).
- Note the difference with mergesort: there, the division of the problem into two subproblems 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 subproblems.
- Here is pseudocode of quicksort: call Quicksort(A[0..n − 1]) where



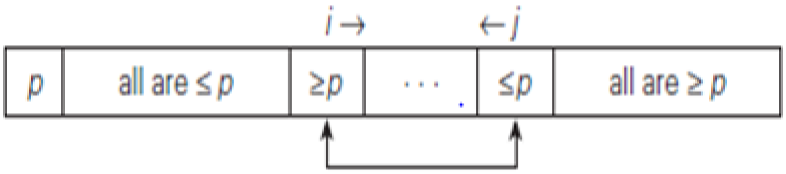
```
ALGORITHM Quicksort(A[l..r])
    //Sorts a subarray by quicksort
    //Input: Subarray of array A[0..n − 1], defined by its left and right
    //       indices l and r
    //Output: Subarray A[l..r] sorted in nondecreasing order
    if l < r
        s ← Partition(A[l..r])  //s is a split position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

Partitioning

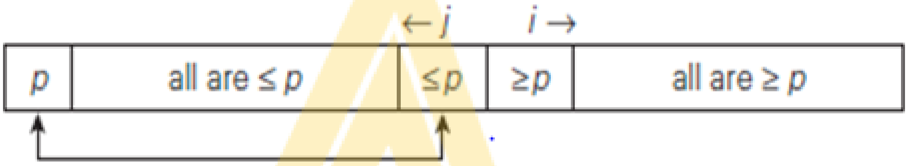
- We start by selecting a pivot—an element with respect to whose value we are going to divide the subarray. There are several different strategies for selecting a pivot wher in here we use the simplest strategy of selecting the subarray's first element: p = A[l].
- We will now scan the subarray from both ends, comparing the subarray's elements to the pivot.
    - ❖ The left-to-right scan, denoted below by index pointer i, starts with the second element. Since we want elements smaller than the pivot to be in the left part of the subarray, this scan skips over elements that are smaller than the pivot and stops upon encountering the first element greater than or equal to the pivot.
    - ❖ The right-to-left scan, denoted below by index pointer j, starts with the last element of the subarray. Since we want elements larger than the pivot to be in the right part

      of the subarray, this scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot.
- After both scans stop, three situations may arise, depending on whether or not the scanning indices have crossed.

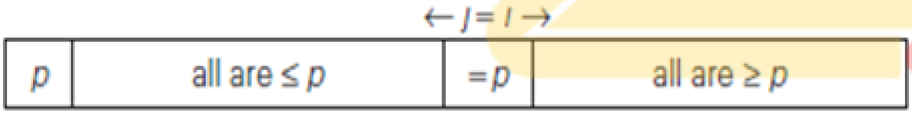
- If scanning indices i and j have not crossed, i.e., i < j, we simply exchange A[i] and A[j] and resume the scans by incrementing i and decrementing j, respectively:



- If the scanning indices have crossed over, i.e., i > j, we will have partitioned the subarray after exchanging the pivot with A[j]:



- 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 .

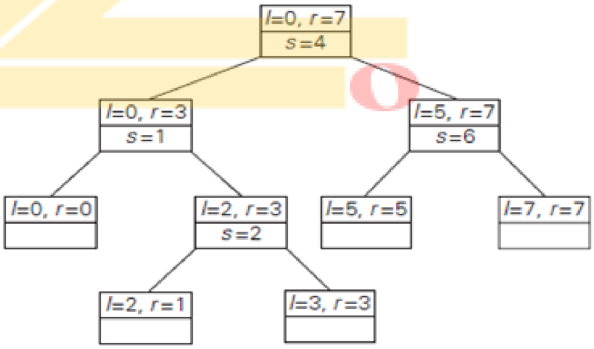


Here is pseudocode implementing this partitioning procedure.

Example: Example of quicksort operation. (a) Array's transformations with pivots shown in bold. (b) Tree of recursive calls to Quicksort with input values l and r of subarray bounds and split position s of a partition obtained

```
ALGORITHM  HoarePartition(A[l..r])
    //Partitions a subarray by Hoare's algorithm, using the first element
    //          as a pivot
    //Input: Subarray of array $A[0..n-1]$, defined by its left and right
    //          indices $l$ and $r$ ($l < r$)
    //Output: Partition of $A[l..r]$, with the split position returned as
    //          this function's value
    $p \leftarrow A[l]$
    $i \leftarrow l$; $j \leftarrow r+1$
    repeat
        repeat $i \leftarrow i+1$ until $A[i] \geq p$
        repeat $j \leftarrow j-1$ until $A[j] \leq p$
        swap($A[i]$, $A[j]$)
    until $i \geq j$
    swap($A[i]$, $A[j]$)    //undo last swap when $i \geq j$
    swap($A[l]$, $A[j]$)
    return $j$
```

0	1	2	3	4	5	6	7
5	$\overset{i}{3}$	1	9	8	2	4	$\overset{j}{7}$
5	3	1	$\overset{i}{9}$	8	2	$\overset{j}{4}$	7
5	3	1	$\overset{i}{4}$	8	2	$\overset{j}{9}$	7
5	3	1	4	$\overset{i}{8}$	$\overset{j}{2}$	9	7
5	3	1	4	$\overset{i}{2}$	$\overset{j}{8}$	9	7
5	3	1	4	$\overset{j}{2}$	$\overset{i}{8}$	9	7
2	3	1	4	**5**	8	9	7
2	$\overset{i}{3}$	1	$\overset{j}{4}$				
2	$\overset{i}{3}$	$\overset{j}{1}$	4				
2	$\overset{i}{1}$	$\overset{j}{3}$	4				
2	$\overset{j}{1}$	$\overset{i}{3}$	4				
1	**2**	3	4				
1							
		3	$\overset{i\,j}{4}$				
		$\overset{j}{\mathbf{3}}$	$\overset{i}{4}$				
			4				
					8	$\overset{i}{9}$	$\overset{j}{7}$
					8	$\overset{i}{7}$	$\overset{j}{9}$
					8	$\overset{j}{7}$	$\overset{i}{9}$
					7	**8**	9
					7		
							9



(b)

Analysis

Best

Case

- Here the basic operation is key comparison. Number of key comparisons made before a partition is achieved is n + 1 if the scanning indices cross over and n if they coincide.
- If all the splits happen in the middle of corresponding subarrays, we will have the best case.
- The number of key comparisons in the best case satisfies the recurrence,
- According to the Master Theorem, $\boldsymbol{C_{best}(n) \in \Theta(n \log_2 n)}$; solving it exactly for n = 2^k yields

$C_{best}(n) = n \log_2 n$.

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. This unfortunate situation will happen, in particular, for increasing arrays, i.e., for inputs for which the problem is already solved
- Indeed, if A[0..n − 1] is a strictly increasing array and we use A[0] as the pivot, the left-to- right scan will stop on A[1] while the right-to-left scan will go all the way to reach A[0], indicating the split at position 0: 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. This sorting of strictly increasing arrays of diminishing sizes will continue until the last one A[n−2 .. n−1] has been processed. The total number of key comparisons made will be equal to

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2).$$

Average Case

- Let $C_{avg}(n)$ 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:

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$$
$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

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

$$C_{avg}(n) \approx 2n \ln n \approx 1.39n \log_2 n.$$

- 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. This certainly justifies the name given to the algorithm by its inventor.

Variations

Because of quicksort‘s importance, there have been persistent efforts over the years to refine the basic algorithm. Among several improvements discovered by researchers are:

- Better pivot selection methods such as ***randomized quicksort*** that uses a random element or the ***median-of-three*** method that uses the median of the leftmost, rightmost, and the middle element of the array
- Switching to insertion sort on very small subarrays (between 5 and 15 elements for most computer systems) or not sorting small subarrays at all and finishing the algorithm with insertion sort applied to the entire nearly sorted array
- Modifications of the partitioning algorithm such as the three-way partition into segments smaller than, equal to, and larger than the pivot

Limitations

- It is not stable.
- It requires a stack to store parameters of subarrays that are yet to be sorted.
- While performance on randomly ordered arrays is known to be sensitive not only to implementation details of the algorithm but also to both computer architecture and data type.

3. Strassen's Matrix

Multiplication Direct Method

- Suppose we want to multiply two n x n matrices, A and B. Their product

C=AB, will be an n by n matrix and will therefore have n^2 elements.

- The number of multiplications involved in producing the product in this way is Θ(n^3)

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

Divide and Conquer method

- Multiplication of 2 × 2 matrices **:** By using divide-and-conquer approach we can reduce the number of multiplications. Such an algorithm was published by V. Strassen in 1969.
- The principal insight of the algorithm lies in the discovery that we can find the product C of two 2× 2 matrices A and B with **just seven multiplications** as opposed to the eight required by the brute-force algorithm.
- This is accomplished by using the following formulas:

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix},$$

$$\begin{aligned}
m_1 &= (a_{00} + a_{11}) * (b_{00} + b_{11}),\\
m_2 &= (a_{10} + a_{11}) * b_{00},\\
m_3 &= a_{00} * (b_{01} - b_{11}),\\
m_4 &= a_{11} * (b_{10} - b_{00}),\\
m_5 &= (a_{00} + a_{01}) * b_{11},\\
m_6 &= (a_{10} - a_{00}) * (b_{00} + b_{01}),\\
m_7 &= (a_{01} - a_{11}) * (b_{10} + b_{11}).
\end{aligned}$$

where

- Thus, to multiply two 2×2 matrices, Strassen's algorithm makes seven multiplications and 18 additions/subtractions, whereas the brute-force algorithm requires eight multiplications and four additions.

Multiplication of n × n matrices – Let A and B be two n × n matrices where n is a power of 2. (If n is not a power of 2, matrices can be padded with rows and columns of zeros.) We can divide A, B, and their product C into four n/2 × n/2 submatrices each as follows:

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] * \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right]$$

It is not difficult to verify that one can treat these submatrices as numbers to get the correct product. For example, C_{00} can be computed either as $A_{00} * B_{00} + A_{01} * B_{10}$ or as $M_1 + M_4 - M_5 + M_7$ where M_1, M_4, M_5, and M_7 are found by Strassen's formulas, with the numbers replaced by the corresponding submatrices. If the seven products of n/2 × n/2 matrices are computed recursively by the same method, we have

Strassen‘s algorithm for matrix multiplication.

Analysis

- Here the basic operation is *multiplication.*
- If M(n) is the number of multiplications made by Strassen‘s algorithm in multiplying two n× n matrices (where n is a power of 2), we get the following recurrence relation for it

$$M(n) = 7M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.$$

Since $n = 2^k$,

$$M(2^k) = 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2 M(2^{k-2}) = \cdots$$
$$= 7^i M(2^{k-i}) \cdots = 7^k M(2^{k-k}) = 7^k.$$

Since $k = \log_2 n$,

$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807},$$

This implies ***$M(n) = \Theta(n^{2.807})$*** which is smaller than n^3 required by the brute-force algorithm.

4. Advantages and Disadvantages of Divide & Conquer

Advantages

- **Parallelism:** Divide and conquer algorithms tend to have a lot of inherent parallelism. Once the division phase is complete, the sub-problems are usually independent and can therefore be solved in parallel. This approach typically generates more enough concurrency to keep the machine busy and can be adapted for execution in multi- processor machines.
- **Cache Performance:** divide and conquer algorithms also tend to have good cache performance. Once a sub-problem fits in the cache, the standard recursive solution

reuses the cached data until the sub-problem has been completely solved.

- It allows us to solve difficult and often impossible looking problems, such as the Tower of Hanoi, which is a mathematical game or puzzle. Being given a difficult problem can often be discouraging if there is no idea how to go about solving it. However, with the divide and conquer method, it reduces the degree of difficulty since it divides the problem into sub problems that are easily solvable, and usually runs faster than other algorithms would.

Disadvantages

- Another advantage to this paradigm is that it often **plays a part in finding other efficient algorithms**, and in fact it was the central role in finding the quick sort and merge sort algorithms.
- One of the most common issues with this sort of algorithm is the fact that the recursion is slow, which in some cases outweighs any advantages of this divide and conquer process.
- Sometimes it can become more complicated than a basic iterative approach, especially in cases with a large n. In other words, if someone wanted to add a large amount of numbers together, if they just create a simple loop to add them together, it would turn out to be a much simpler approach than it would be to divide the numbers up into two groups, add these groups recursively, and then add the sums of the two groups together.
- Another downfall is that sometimes once the problem is broken down into sub problems, the same sub problem can occur many times. In cases like these, it can often be easier to identify and save the solution to the repeated sub problem, which is commonly referred to as memorization

5. DECREASE AND CONQUER APPROACH

5.1 Introduction

Decrease & conquer is a general algorithm design strategy based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem. The exploitation can be either top-down (recursive) or bottom- up (non-recursive).

The major variations of decrease and conquer are

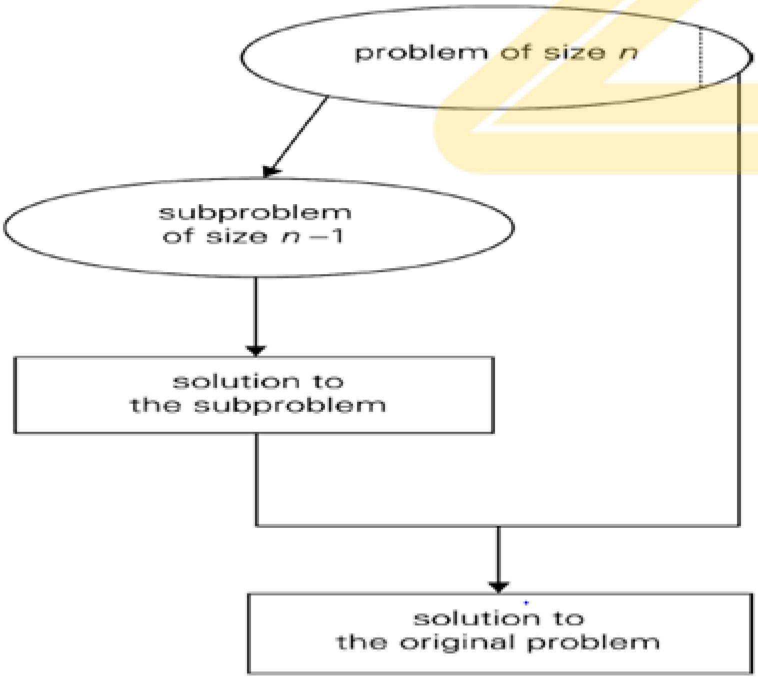
1. Decrease by a constant :(usually by 1): Examples:

    a. Insertion sort
    b. Graph traversal algorithms (DFS and BFS)
    c. Topological sorting
    d. Algorithms for generating permutations, subsets

2. Decrease by a constant factor (usually by half) Example: Binary search
3. Variable size decrease Example: Euclid‘s algorithm

Decrease by a constant :(usually by 1):

In the decrease-by-a-constant variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one (as shown in the figure), although other constant size reductions do happen occasionall.



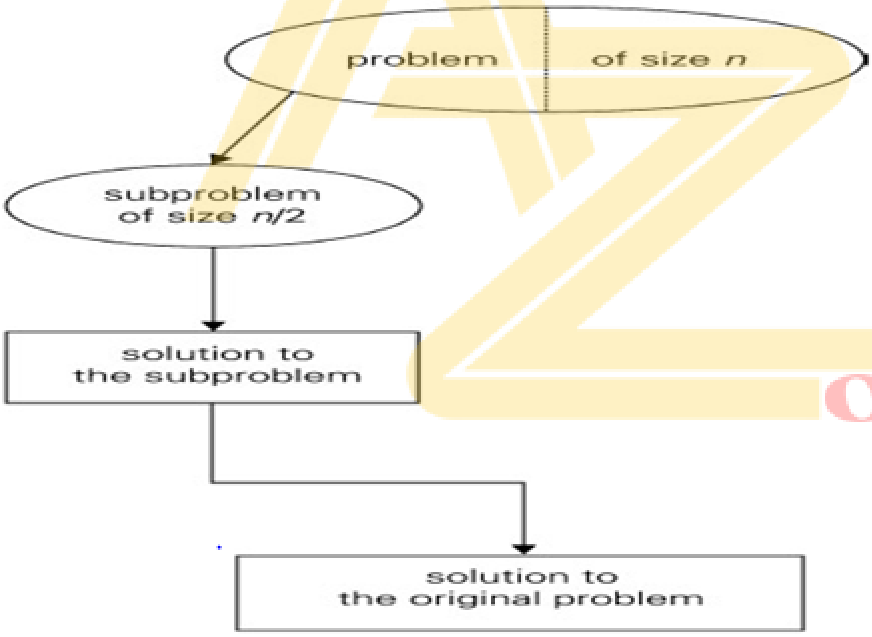
Consider, as an example, the exponentiation problem of computing a^n for positive integer exponents. The relationship between a solution to an instance of size n and an instance of size n-1 is obtained by the obvious formula $a^n = a^{n-1}$. a. So the function f(n) = a^n can be computed either -top down‖ by using its recursive definition

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 0, \\ 1 & \text{if } n = 0, \end{cases}$$

or -bottom up‖ by multiplying a by itself n -1 times.

Decrease by a constant factor :(usually by 2):

The decrease-by-a-constant-factor technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm. In most applications, this constant factor is equal to two. The decrease-by-half idea is illustrated in the following figure:



For an example, let us revisit the exponentiation problem. If the instance of size n is to compute a^n, the instance of half its size is to compute $a^{n/2}$, with the obvious relationship between the two: $a^n = (a^{n/2})^2$.

But since we consider here instances with integer exponents only, the former does not work for odd n. If n is odd, we have to compute a^{n-1} by using the rule for even-valued exponents and then multiply the result by a. To summarize, we have the following formula:

Variable size decrease:

Finally, in the variable-size-decrease variety of decrease-and-conquer, the size-reduction pattern varies from one iteration of an algorithm to another. Euclid‘s algorithm for computing the greatest common divisor provides a good example of such a situation. This algorithm is based on the formula. gcd(m, n) = gcd(n, m mod n).

Topological Sort

Background

A ***directed graph***, or ***digraph*** for short, is a graph with directions specified for all its edges. The adjacency matrix and adjacency lists are still two principal means of representing a digraph.

There are only two notable differences between undirected and directed graphs in representing them: (1) the adjacency matrix of a directed graph does not have to be symmetric; (2) an edge in a directed graph has just one (not two) corresponding nodes in the digraph‘s adjacency lists.

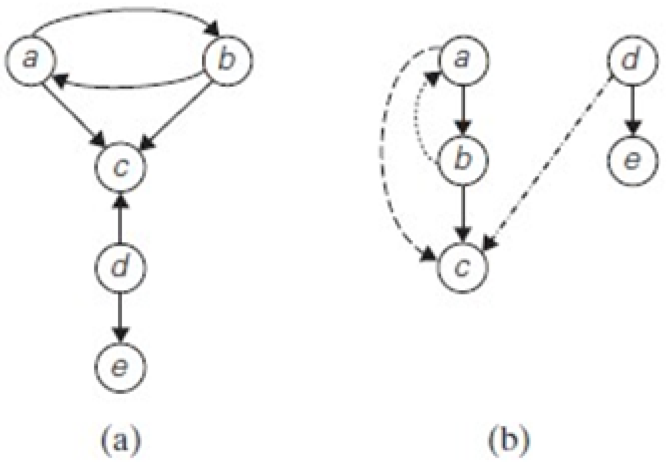
Depth-first search and breadth-first search are principal traversal algorithms for traversing digraphs as well, but the structure of corresponding forests can be more complex than for undirected graphs. Thus, even for the simple example (Figure a) of the depth-first search forest (Figure b) exhibits all four types of edges possible in a DFS forest of a directed graph:

tree edges (*ab*, *bc, de)*

backedges (*ba)* from vertices to their ancestors,

forward edges (*ac)* from vertices to their descendants in the tree other than their children

cross edges (*dc)*, which are none of the aforementioned types.



(a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at a.

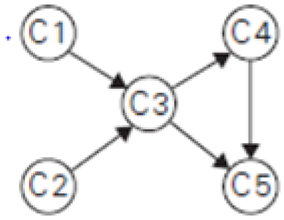
Note that a back edge in a DFS forest of a directed graph can connect a vertex to its parent. Whether or not it is the case, the presence of a back edge indicates that the digraph has a directed cycle. A ***directed cycle*** in a digraph is a sequence of three or more of its vertices that starts and ends with the same vertex and in which every vertex is connected to its immediate predecessor by an edge directed from the predecessor to the successor. For example, *a, b, a* is

a directed cycle in the digraph in Figure a. Conversely, if a DFS forest of a digraph has no back edges, the digraph is a ***dag***, an acronym for ***directed acyclic graph***.

Motivation for topological sorting

- Let us consider a set of five required courses {C1, C2, C3, C4, C5} a part-time student has to take in some degree program.
- The courses can be taken in any order as long as the following course prerequisites are met:
  - C1 and C2 have no prerequisites,
  - C3 requires C1 and C2,
  - C4 requires C3,
  - and C5 requires C3 and C4.
- The student can take only one course per term. In which order should the student take the courses?

- The situation can be modeled by a digraph in which vertices represent courses and directed edges indicate prerequisite requirements (as shown in the Figure).



- In terms of this digraph, the question is whether we can list its vertices in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends. (Can you find such an ordering of this digraph's vertices?) This problem is called ***topological sorting***.
- It can be posed for an arbitrary digraph, but it is easy to see that the problem cannot have a solution if a digraph has a directed cycle.
- Thus, for topological sorting to be possible, a digraph in question must be a directed acyclic graph (dag).
- Two algorithms for obtaining topological sort of a digraph is as follows:
  - ➢ The first algorithm is a simple application of depth-first search:

- perform a DFS traversal and note the order in which vertices become dead-ends (i.e., popped off the traversal stack).
- Reversing this order yields a solution to the topological sorting problem, provided, of course, no back edge has been encountered during the traversal. If a back edge has been encountered, the digraph is not a dag, and topological sorting of its vertices is impossible.

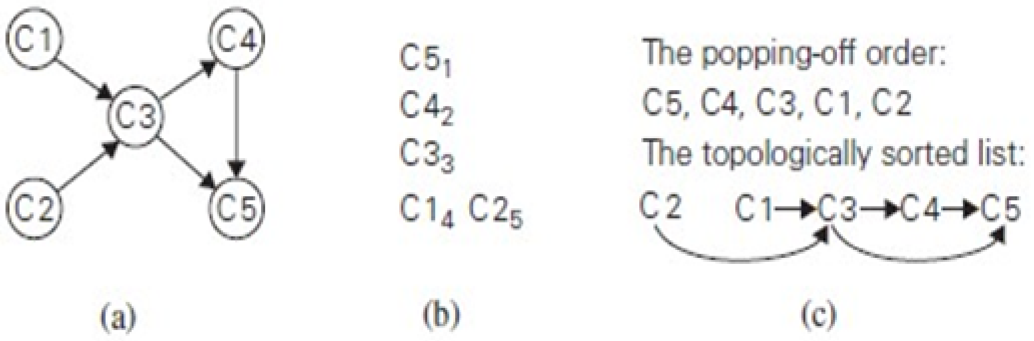
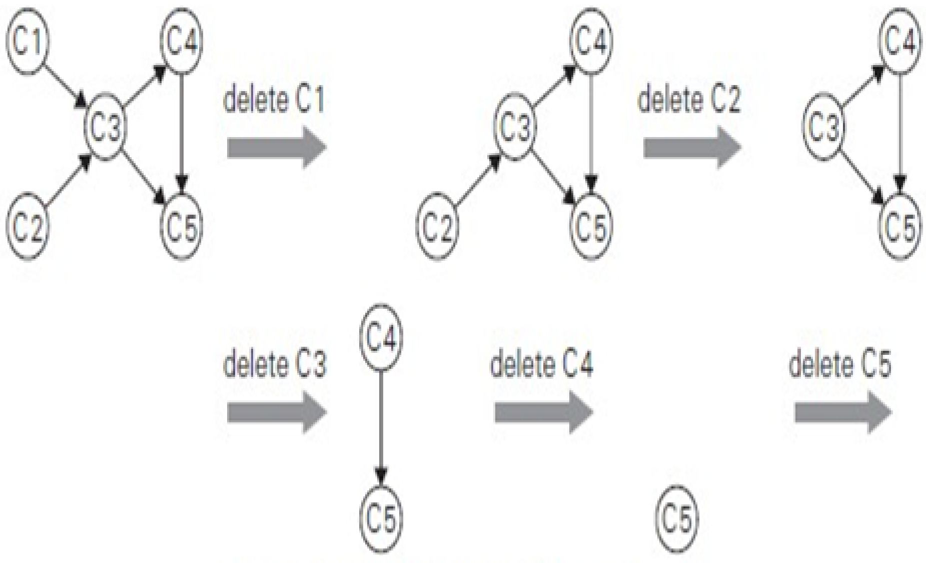


Figure: (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

- The second algorithm is based on a direct implementation of the decrease-(by one)- and-conquer technique:
- repeatedly, identify in a remaining digraph a ***source***, which is a vertex with no incoming edges, and
- delete it along with all the edges outgoing from it. (If there are several sources, break the tie arbitrarily. If there are none, stop because the problem cannot be solved)
- The order in which the vertices are deleted yields a solution to the topological sorting problem.
- The application of this algorithm to the same digraph representing the five courses is given in the following Figure.



The solution obtained is C1, C2, C3, C4, C5

Figure: Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.

Note: The solution obtained by the source-removal algorithm is different from the one obtained by the DFS-based algorithm. Both of them are correct, because the topological sorting problem may have several alternative solutions.

Applications of Topological Sorting

- Instruction scheduling in program

compilation • Cell evaluation ordering in

•spreadsheet formulas,

Resolving symbol dependencies in linkers.