Algorithms
Chapter 3: Decrease and Conquer

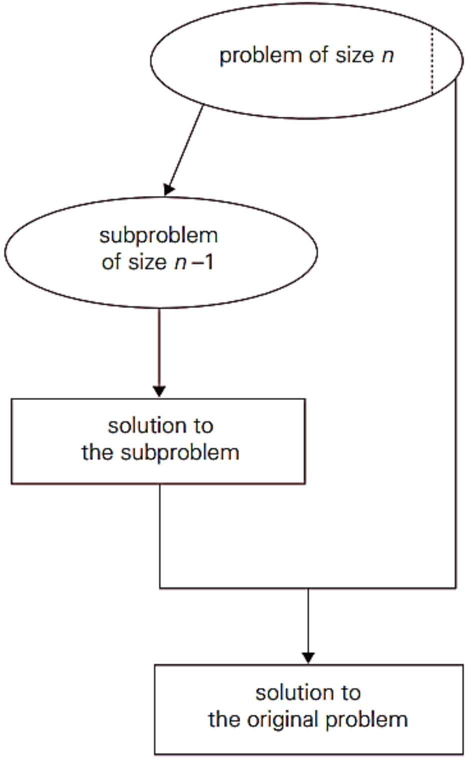
GATE CS Lectures
by Monalisa

Section 5: Algorithms

- Searching, sorting, hashing. Asymptotic worst case time and space complexity. Algorithm design techniques : greedy, dynamic programming and divide-and-conquer . Graph traversals, minimum spanning trees, shortest paths
- Chapter 1:Algorithim Analysis:-Algorithm intro , Order of growth ,Asymptotic notation, Time complexity, space complexity, Analysis of Recursive & non recursive program, Master theorem]
- Chapter 2:Brute Force:-Sequential search, Selection Sort and Bubble Sort , Radix sort, Depth first Search and Breadth First Search.
- Chapter 3: Decrease and Conquer :- Insertion Sort, Topological sort,Binary Search .
- Chapter 4: Divide and conquer:-Min max problem , matrix multiplication ,Merge sort ,Quick Sort , Binary Tree Traversals and Related Properties .
- Chapter 5: Transform and conquer:- Heaps and Heap sort, Balanced Search Trees.
- Chapter 6: Greedy Method:-knapsack problem , Job Assignment problem, Optimal merge, Hoffman Coding, minimum spanning trees, Dijkstra's Algorithm.
- Chapter 7: Dynamic Programming:-The Bellman-Ford algorithm ,Warshall's and Floyd's Algorithm ,Rod cutting, Matrix-chain multiplication ,Longest common subsequence ,Optimal binary search trees
- Chapter 8: Hashing.
- Reference : Introduction to Algorithms by Thomas H. Cormen
- Introduction to the Design and Analysis of Algorithms, by Anany Levitin
- My Note

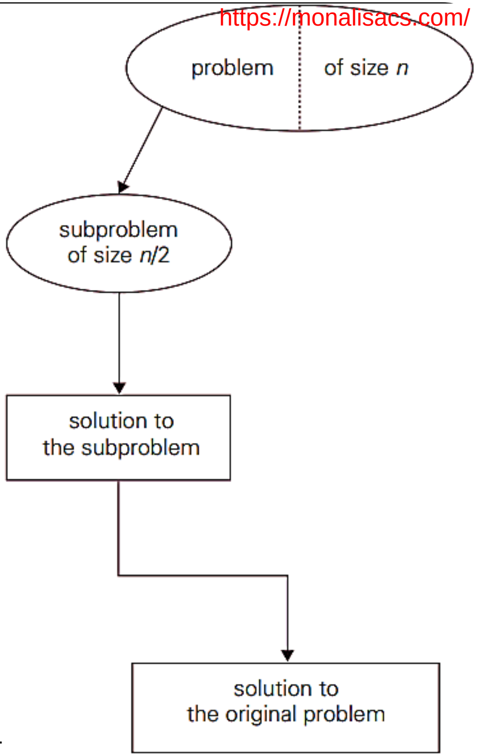
- Chapter 3:
- Decrease and Conquer :-
- Insertion Sort,
- Topological sort,
- Binary Search

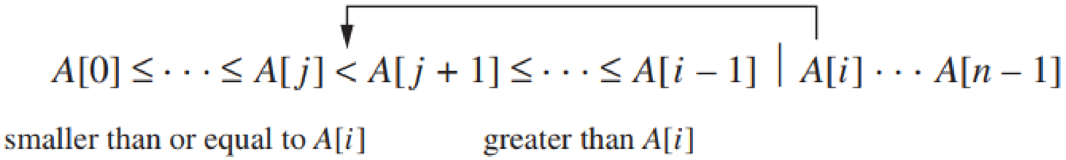
Decrease-and-Conquer



- The ***decrease-and-conquer*** technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instance.
- Once such a relationship is established, it can be exploited either top down or bottom up.
- The bottom-up variation is usually implemented iteratively, starting with a solution to the smallest instance of the problem; it is called sometimes the ***incremental approach***.
- There are three major variations of decrease-and-conquer:
- decrease by a constant
- decrease by a constant factor
- variable size decrease
- 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 , although other constant size reductions do happen occasionally.
- FIGURE : Decrease-(by one)-and-conquer technique

- 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.
- FIGURE 4.2 Decrease-(by half)-and-conquer technique
- 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. gcd(m, n) = gcd$(n, m \bmod n)$
- **Insertion Sort**
- An application of the decrease-by-one technique to sorting an array.
- We assume that the smaller problem of sorting the array $A[0..n - 2]$ has already been solved to give us a sorted array of size $n - 1$.
- All we need is to find an appropriate position for $A[n - 1]$ among the sorted elements and insert it there.
- This is usually done by scanning the sorted subarray from right to left until the first element smaller than or equal to $A[n - 1]$ is encountered to insert $A[n - 1]$ right after that element.





- The algorithm sorts the input numbers **in place**: it rearranges the numbers within the array A, with at most a constant number of them stored outside the array at any time.
- **ALGORITHM** *InsertionSort(A*[0..*n* - 1])
- //Input: An array *A*[0..*n* - 1] of *n* orderable elements
- //Output: Array *A*[0..*n* - 1] sorted in nondecreasing order
- **for** $i \leftarrow 1$ **to** $n - 1$ **do**
- $v \leftarrow A[i]$
- $j \leftarrow i - 1$
- **while** $j \geq 0$ **and** $A[j] > v$ **do**
- $A[j + 1] \leftarrow A[j]$
- $j \leftarrow j - 1$
- $A[j + 1] \leftarrow v$

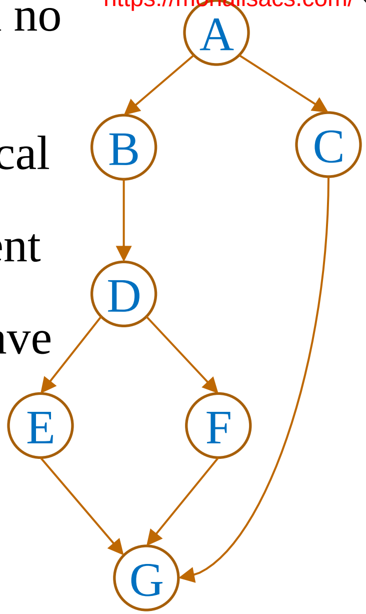
0	1	2	3	4	5	6
8	4	6	9	2	3	1
4	8	6	9	2	3	1
4	6	8	9	2	3	1
4	6	8	9	2	3	1
2	4	6	8	9	3	1
2	3	4	6	8	9	1
1	2	3	4	6	8	9

- The basic operation of the algorithm is the key comparison $A[j] > v$.
- In the worst case, $A[j] > v$ is executed the largest number of times, i.e., for every $j = i - 1, \ldots, 0$.
- $C_{worst}(n) = \sum_{i=1}^{n-1} * \sum_{j=0}^{i-1} 1$
- $= \sum_{i=1}^{n-1} [(i - 1) - 0 + 1] = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2)$
- In the best case, the comparison $A[j] > v$ is executed only once on every iteration of the outer loop.
- For sorted arrays, the number of key comparisons is
- $C_{best}(n) = \sum_{i=1}^{n-1} 1 = n-1 \in \Theta(n)$

Topological Sort

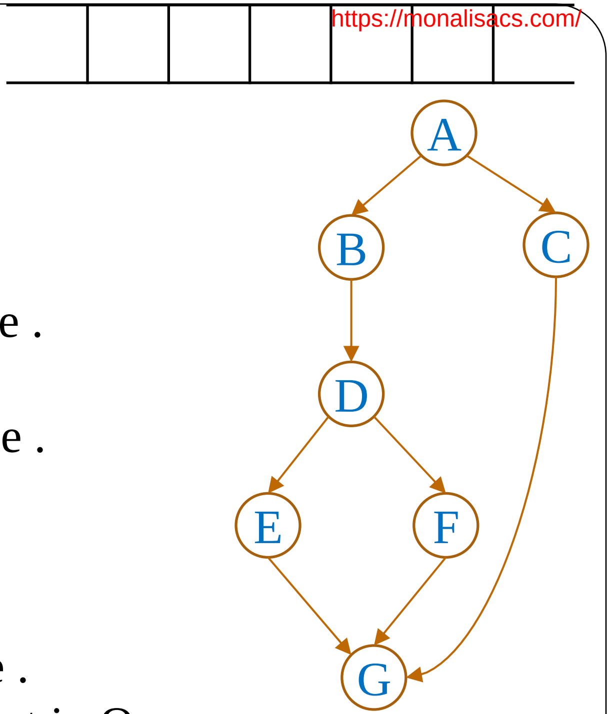
- A ***directed graph***, or ***digraph*** , is a graph with directions specified for all its edges.
- Four types of edges possible in a DFS forest of a directed graph: ***tree edges*** , ***back edges*** from vertices to their ancestors, ***forward edges*** from vertices to their descendants in the tree other than their children, and ***cross edges*** , which are none of the back edges or forward edges.
- If a DFS forest of a digraph has no back edges, the digraph is a ***DAG***, an acronym for ***directed acyclic graph*** .
- A ***topological sort*** of a DAG G=(V,E) is a linear ordering of all its vertices such that for every directed edge (u,v) then u appears before v in the ordering.
- If the graph contains a cycle , then no linear ordering is possible.
- We can view a topological sort of a graph as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.
- TOPOLOGICAL-SORT(G)
- 1. call DFS(G) to compute finishing times *v.f* for each vertex .
- 2. as each vertex is finished, insert it onto the front of a linked list.
- 3. **return** the linked list of vertices
- We can perform a topological sort in time Θ (V+E), Since depth-first search takes Θ (V+ E) time and it takes O(1) time to insert each of the |V| vertices onto the front of the linked list.
- 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.)
- The order in which the vertices are deleted yields a solution to the topological sorting problem .
- Note that the solution obtained by the **source-removal algorithm** is different from the one obtained by the DFS-based algorithm.
- Both of them are correct, of course; the topological sorting problem may have several alternative solutions.
- Example 1:

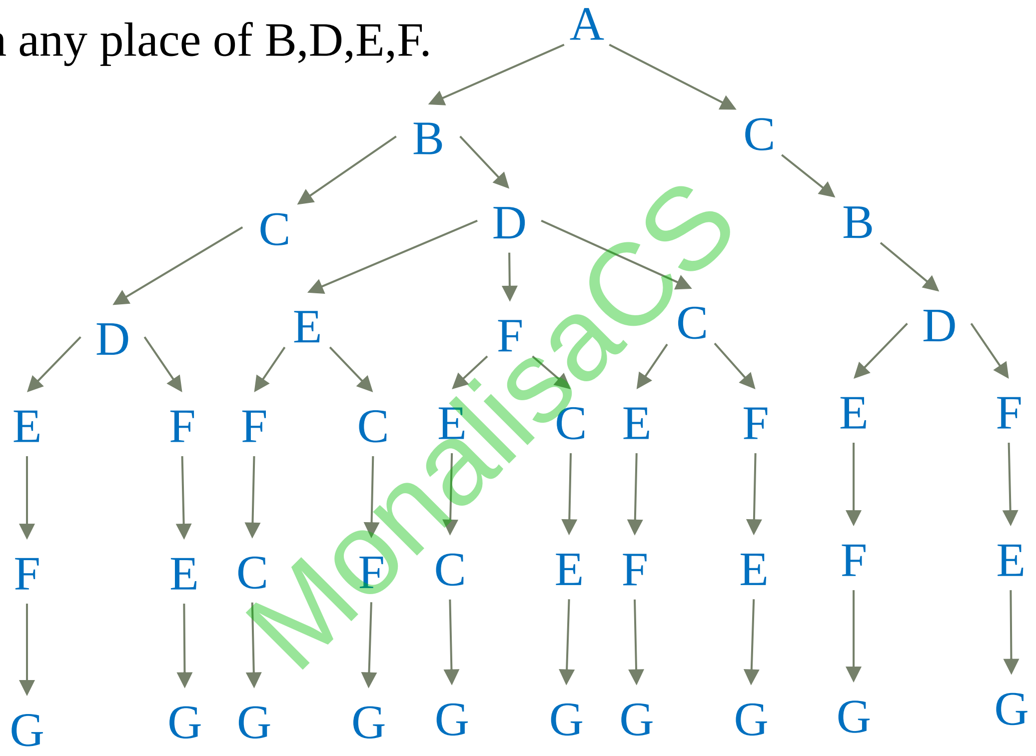


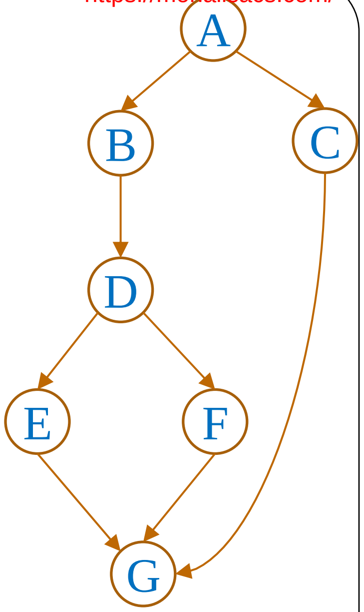
- DFS Sequence 1: $A_{1/14}$, $B_{2/11}$, $D_{3/10}$, $E_{4/7}$, $G_{5/6}$, $F_{8/9}$, $C_{12/13}$
- Topological sort Sequence 1: A , C , B , D , F , E , G
- DFS Sequence 2: $A_{1/14}$, $B_{2/11}$, $D_{3/10}$, $F_{4/7}$, $G_{5/6}$, $E_{8/9}$, $C_{12/13}$
- Topological sort Sequence 2: A , C , B , D , E , F , G
- DFS Sequence 3: $A_{1/14}$, $C_{2/5}$, $G_{3/4}$, $B_{6/13}$, $D_{7/12}$, $E_{8/9}$, $F_{10/11}$
- Topological sort Sequence 3: A , B , D , F , E , C , G
- DFS Sequence 4: $A_{1/14}$, $C_{2/5}$, $G_{3/4}$, $B_{6/13}$, $D_{7/12}$, $F_{8/9}$, $E_{10/11}$
- Topological sort Sequence 4: A , B , D , E , F , C , G
- Number of Different Topological ordering =4

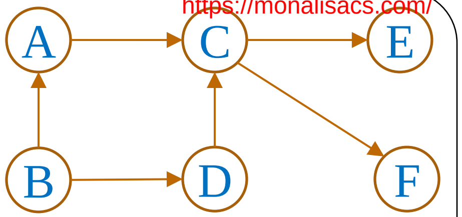
- 2^{nd} way : **source-removal algorithm**
- Find indegree of all vertices and write in increasing order .
- (A,0) , (B,1) , (C,1) , (D,1) , (E,1) , (F,1) , (G,3)
- Remove A with its outgoing edge and insert in queue
- Indegree: (B,0) , (C,0) , (D,1) , (E,1) , (F,1) , (G,3)
- Remove B or C..let B with its outgoing edge and insert in Queue .
- Indegree: (C,0) , (D,0) , (E,1) , (F,1) , (G,3)
- Remove C or D..let C with its outgoing edge and insert in Queue .
- Indegree (D,0) , (E,1) , (F,1) , (G,2)
- Remove D with its outgoing edge and insert in Queue .
- Indegree (E,0) , (F,0) , (G,2)
- Remove E or F..let E with its outgoing edge and insert in Queue .
- Indegree: (F,0),(G,1) ,Remove F with its outgoing edge and insert in Queue .
- Indegree: (G,0),Remove G and insert in Queue .
- TS 1:A,B,C,D,E,F,G TS 2:A,B,C,D,F,E,G TS 3:A,B,D,E,C,F,G
- TS 4:A,B,D,E,F,C,G TS 5:A,B,D,F,E,C,G TS 6:A,B,D,F,C,E,G
- TS 7:A,B,D,C,F,E,G TS 8:A,B,D,C,E,F,G TS 9:A,C,B,D,E,F,G
- TS 10:A,C,B,D,F,E,G Number of Different Topological ordering =10



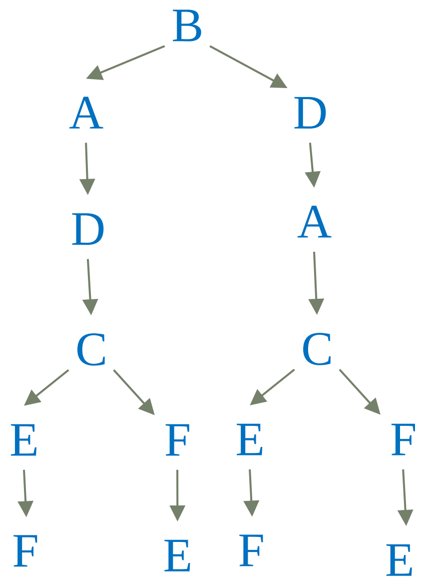
- A_ _ _ _ _ G
- B come before D , D come before E or F.
- C can come between any place of B,D,E,F.
- $\frac{5!}{\frac{4!}{2!}\times 1!}=10$







- Example 2:
- DFS1:$B_{1/12}$,$A_{2/9}$,$C_{3/8}$,$E_{4/5}$,$F_{6/7}$,$D_{10/11}$, TS1: B ,D ,A, C, F, E
- DFS2:$B_{1/12}$,$A_{2/9}$,$C_{3/8}$,$F_{4/5}$,$E_{6/7}$,$D_{10/11}$, TS2: B ,D ,A, C, E, F
- DFS3:$B_{1/12}$,$D_{2/9}$,$C_{3/8}$,$F_{4/5}$,$E_{6/7}$,$A_{10/11}$, TS3: B ,A ,D, C, E, F
- DFS4:$B_{1/12}$,$D_{2/9}$,$C_{3/8}$,$E_{4/5}$,$F_{6/7}$,$A_{10/11}$, TS4: B ,A ,D, C, F, E
- Number of Different Topological ordering =4
- 2nd way : **source-removal algorithm**
- Indegree:(B,0),(A,1),(D,1),(E,1),(F,1),(C,2)
- Remove B with its outgoing edge and insert in Queue .
- Indegree:(A,0),(D,0),(E,1),(F,1),(C,2), Remove A or D let A .
- Indegree:(D,0),(E,1),(F,1),(C,1), Remove D .
- Indegree:(C,0),(E,1),(F,1), Remove C .
- Indegree:(E,0),(F,0), Remove E or F let E.
- Remove F and insert in Queue .
- Topological sort 1:B,A,D,C,E,F ,Topological sort 2:B,A,D,C,F,E
- Topological sort 3:B,D,A,C,E,F, Topological sort 4:B,D,A,C,F,E



Decrease-by-a-Constant-Factor Algorithms

- Decrease-by-a-constant-factor algorithms usually run in logarithmic time.

➢ Binary Search

- Binary search is a algorithm for searching in a sorted array.
- It works by comparing a search key K with the array's middle element $A[m]$.
- If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if $K < A[m]$, and for the second half if $K > A[m]$:
- **ALGORITHM** *BinarySearch(A[0..n - 1], K)*
- //Input: An array $A[0..n-1]$ sorted in ascending order and a search key K.
- //Output: An index of the array's element that is equal to K or -1 if there is no such element
- $l \leftarrow 0;\ r \leftarrow n-1$
- **while** $l \le r$ **do**
- $m \leftarrow \lfloor (l+r)/2 \rfloor$
- **if** $K = A[m]$ **return** m
- **else if** $K < A[m]$ $r \leftarrow m-1$
- **else** $l \leftarrow m+1$
- **return** -1
- The worst-case inputs include all arrays that do not contain a given search key, as well as some successful searches.

$$\begin{array}{c} K \\ \updownarrow \\ \underbrace{A[0]\ldots A[m-1]}_{\text{search here if } K<A[m]}\quad A[m]\quad \underbrace{A[m+1]\ldots A[n-1]}_{\text{search here if } K>A[m]}. \end{array}$$

- Since after one comparison the algorithm faces the same situation but for an array half the size , we get the following recurrence relation for $C_{worst}(n)$:
- $C_{worst}(n) = C_{worst}\lfloor n/2 \rfloor + 1$ for $n > 1$, $C_{worst}(1) = 1$.
- T(n)=T(n/2)+1
- a=1,b=2,f(n)=1
- $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$
- *Case 2 :* $f(n)=\Theta(n^{\log_b a})$ *then* $T(n)$ *is* $\Theta(n^{\log_b a} * \log n)$
- $T(n)$ *is* $\Theta(\log n)$
- $C_{worst}(n) = \log_2 n + 1$. $C_{Best}(n) = 1$
- As an example, let us apply binary search to searching for $K = 70$ in the array

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
1	*l*						*m*						*r*
2								*l*		*m*			*r*
3								*l,m*	*r*				

- **Decrease-and-Conquer Recurrence**
- **Master Theorem for Decrease & conquer Recurrence**
- *T(n)=aT(n-b)+f(n)* [a>0 ,b>0 ,T(d)=c Initial condition ,n>d]
- *Case 1:if a<1,* T(n) is O(f(n))
- *Case 2:if a=1,* T(n) is O(n*f(n))
- *Case 3:if a>1,* T(n) is $O(a^{n/b} * f(n))$
- **Ex 1:** T(n)=T(n-1)+1
- **Ex 2:** T(n)=T(n-1)+n
- **Ex 3:** T(n)=T(n-1)+log n
- **Ex 4:** T(n)=n*T(n-1)+1
- **Ex 5:** T(n)=2T(n-1)+1
- **Ex 6:** T(n)=2T(n-1)+n
- **Ex 7:** T(n)=1/2T(n-1)+log n