UNIT-5
DECREASE-AND-CONQUER APPROACHES, SPACE-TIMETRADEOFFS

5.1 Decrease and conquer :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

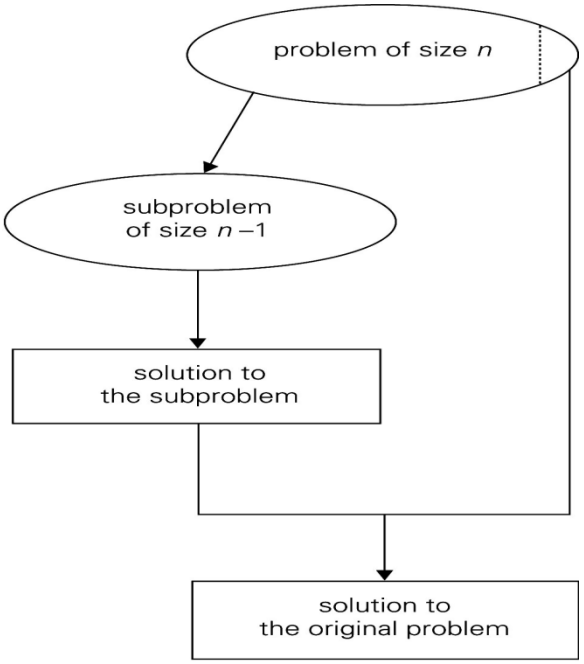
1. Decrease by a constant :(usually by 1):
    a. insertion sort
    b. graph traversal algorithms (DFS and BFS)
    c. topological sorting
    d. algorithms for generating permutations, subsets

2. Decrease by a constant factor (usually by half)
    a. binary search and bisection method

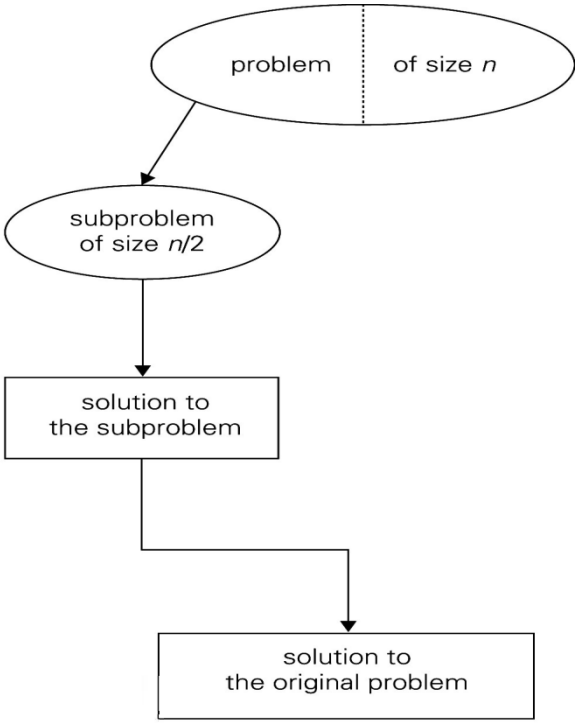
3. Variable size decrease
    a. Euclid‘s algorithm

Following diagram shows the major variations of decrease & conquer approach.

Decrease by a constant :(usually by 1):


problem of size n
subproblem of size n –1
solution to the subproblem
solution to the original problem

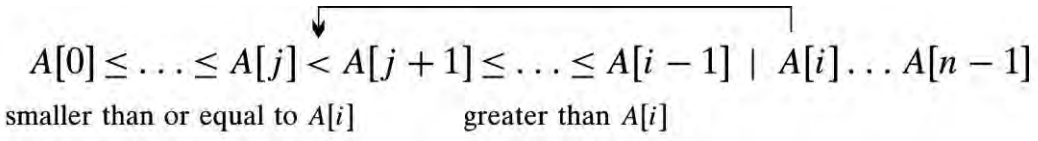
Decrease by a constant factor (usually by half)



5.2 Insertion sort

Description:

Insertion sort is an application of decrease & conquer technique. It is a comparison based sort in which the sorted array is built on one entry at a time



Algorithm:

ALGORITHM Insertionsort(A [0 … n-1])
//sorts a given array by insertion sort
//i/p: Array A[0…n-1]
//o/p: sorted array A[0…n-1] in ascending order

```
for i    →   1 to n-1
               V→     A[i]
               j→     i-1
               while j ≥ 0 AND A[j] > V do
                        A[j+1]     A[j]
                        j→     j – 1
               A[j + 1]→      V
```

Analysis:

- **Input size:** Array size, n
- **Basic operation:** key comparison
- Best, worst, average case exists
  Best case: when input is a sorted array in ascending order:

Worst case: when input is a sorted array in descending order:

- Let $C_{worst}(n)$ be the number of key comparison in the worst case. Then

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2).$$

- Let $C_{best}(n)$ be the number of key comparison in the best case. Then

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

Example:

Sort the following list of elements using insertion sort:

89, 45, 68, 90, 29, 34, 17

89	**45**	68	90	29	34	17
45	89	**68**	90	29	34	17
45	68	89	**90**	29	34	17
45	68	89	90	**29**	34	17
29	45	68	89	90	**34**	17
29	34	45	68	89	90	**17**
17	29	34	45	68	89	90

Advantages of insertion sort:

- Simple implementation. There are three variations
  - Left to right scan
  - Right to left scan

- Binary insertion sort
- Efficient on small list of elements, on almost sorted list
- Running time is linear in best case
- Is a stable algorithm
- Is a in-place algorithm

5.3 Depth-first search (DFS) and Breadth-first search (BFS)

DFS and BFS are two graph traversing algorithms and follow decrease and conquer approach – decrease by one variation to traverse the graph

Some useful definition:

- **Tree edges:** edges used by DFS traversal to reach previously unvisited vertices
- **Back edges:** edges connecting vertices to previously visited vertices other than their immediate predecessor in the traversals
- **Cross edges:** edge that connects an unvisited vertex to vertex other than its immediate predecessor. (connects siblings)
- **DAG:** Directed acyclic graph

Depth-first search (DFS)

Description:

- DFS starts visiting vertices of a graph at an arbitrary vertex by marking it as visited.
- It visits graph‘s vertices by always moving away from last visited vertex to an unvisited one, backtracks if no adjacent unvisited vertex is available.
- Is a recursive algorithm, it uses a stack
- A vertex is pushed onto the stack when it‘s reached for the first time
- A vertex is popped off the stack when it becomes a dead end, i.e., when

there is no adjacent unvisited vertex

- –Redraws" graph in tree-like fashion (with tree edges and back edges for undirected graph)

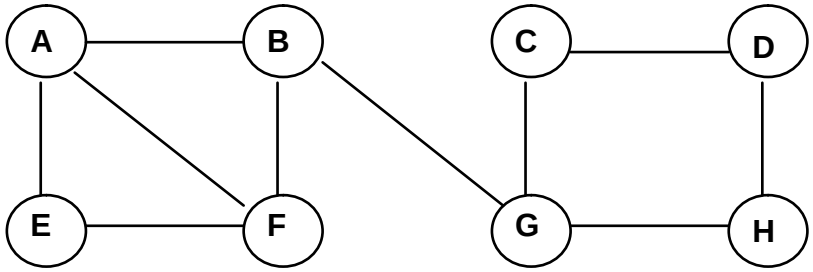
Algorithm:

ALGORITHM DFS (G)

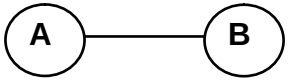
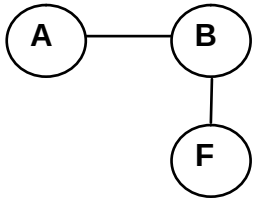
```
//implements DFS traversal of a given graph
//i/p: Graph G = { V, E}
//o/p: DFS tree
Mark each vertex in V with 0 as a mark of being –unvisited"
count→       0
for each vertex v in V do
        if v is marked with 0
                dfs(v)
dfs(v)
count→      count + 1
mark v with count
for each vertex w in V adjacent to v do
        if w is marked with 0
                dfs(w)
```

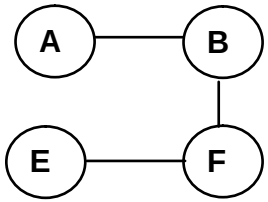
Example:

Starting at vertex A traverse the following graph using DFS traversal method:



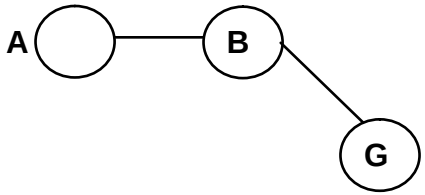
Solution:

Step	Graph	Remarks
1		Insert A into stack A(1)
2		Insert B into stack B (2) A(1)
3		Insert F into stack F (3) B (2) A(1)
4		

		Insert E into stack E (4) F (3) B (2) A(1)
5	NO unvisited adjacent vertex for E, backtrack	Delete E from stack E (4, 1) F (3) B (2) A(1)

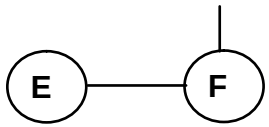
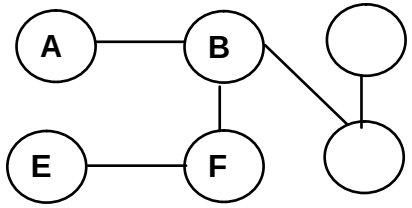
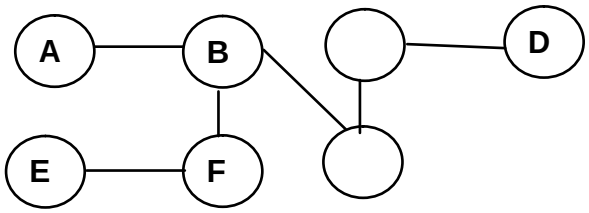
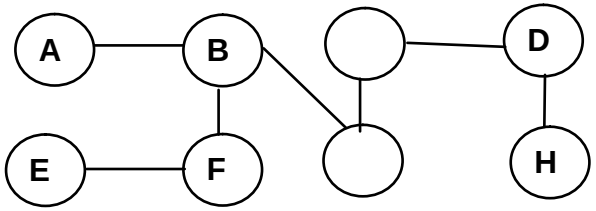
6	NO unvisited adjacent vertex for F, backtrack	Delete F from stack E (4, 1) F (3, 2) B (2) A(1)

7



Insert G into stack

8

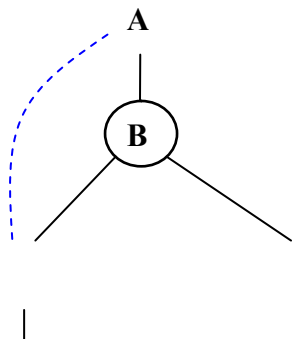
		E (4, 1) F (3, 2) G (5) B (2) A(1)
		Insert C into stack E (4, 1) C (6) F (3, 2) G (5) B (2) A(1)
		Insert D into stack D (7) E (4, 1) C (6) F (3, 2) G (5) B (2) A(1)
10		Insert H into stack H (8) D (7) E (4, 1) C (6) F (3, 2) G (5) B (2) A(1)
11	NO unvisited adjacent vertex for H, backtrack	Delete H from stack

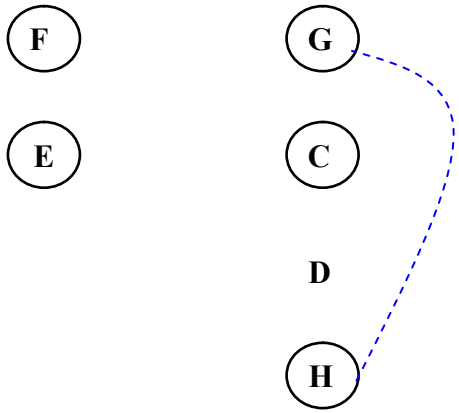
		H (8, 3) D (7) E (4, 1) C (6) F (3, 2) G (5) B (2) A(1)

12	NO unvisited adjacent vertex for D, backtrack	Delete D from stack H (8, 3) D (7, 4) E (4, 1) C (6) F (3, 2) G (5) B (2) A(1)
13	NO unvisited adjacent vertex for C, backtrack	Delete C from stack H (8, 3) D (7, 4) E (4, 1) C (6, 5) F (3, 2) G (5) B (2) A(1)
14	NO unvisited adjacent vertex for G, backtrack	Delete G from stack H (8, 3) D (7, 4) E (4, 1) C (6, 5) F (3, 2) G (5, 6) B (2)

		A(1)
15	NO unvisited adjacent vertex for B, backtrack	Delete B from stack H (8, 3) D (7, 4) E (4, 1) C (6, 5) F (3, 2) G (5, 6) B (2, 7) A(1)
16	NO unvisited adjacent vertex for A, backtrack	Delete A from stack H (8, 3) D (7, 4) E (4, 1) C (6, 5) F (3, 2) G (5, 6) B (2, 7) A(1, 8)
	Stack becomes empty. Algorithm stops as all the nodes in the given graph are visited	

The DFS tree is as follows: (dotted lines are back edges)





Applications of DFS:

- The two orderings are advantageous for various applications like topological sorting, etc
- To check connectivity of a graph (number of times stack becomes empty tells the number of components in the graph)
- To check if a graph is acyclic. (no back edges indicates no cycle)
- To find articulation point in a graph

Efficiency:

- Depends on the graph representation:
  - Adjacency matrix : $\Theta(n^2)$
  - Adjacency list: $\Theta(n + e)$

Breadth-first search (BFS)

Description:

- BFS starts visiting vertices of a graph at an arbitrary vertex by marking it

as visited.

- It visits graph's vertices by across to all the neighbors of the last visited vertex
- Instead of a stack, BFS uses a queue
- Similar to level-by-level tree traversal
- "Redraws" graph in tree-like fashion (with tree edges and cross edges for undirected graph)

Algorithm:

ALGORITHM BFS (G)

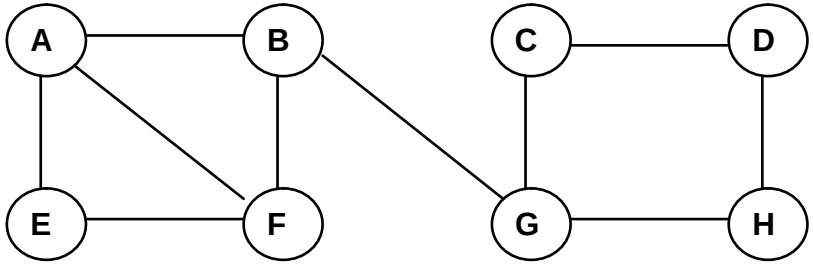
```
//implements BFS traversal of a given graph
//i/p: Graph G = { V, E}
//o/p: BFS tree/forest
Mark each vertex in V with 0 as a mark of being "unvisited"
count→ 0
for each vertex v in V do
    if v is marked with 0
        bfs(v)
```

bfs(v)

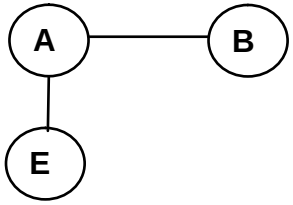
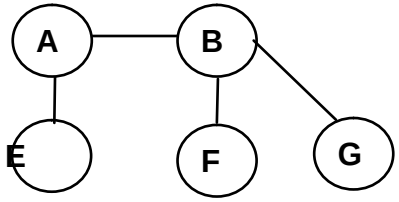
```
count→ count + 1
mark v with count and initialize a queue with v
while the queue is NOT empty do
    for each vertex w in V adjacent to front's vertex v do
        if w is marked with 0
            count→ count + 1
            mark w with count
            add w to the queue
        remove vertex v from the front of the queue
```

Example:

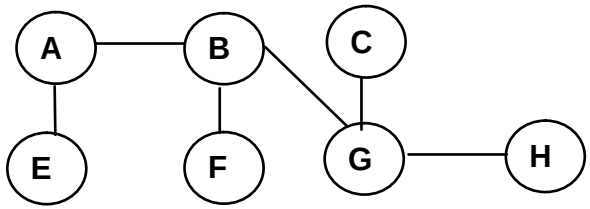
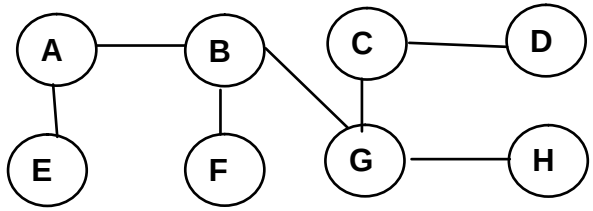
Starting at vertex A traverse the following graph using BFS traversal method:



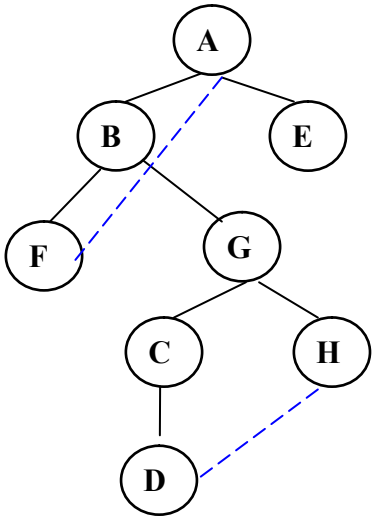
Solution:

Step	Graph	Remarks
1		Insert A into queue A(1)
2		Insert B, E into queue A(1), B (2), E(3) B (2), E(3)
3		Insert F, G into queue B(2), E(3), F(3), G(4)

E(3), F(3), G(4)

4	NO unvisited adjacent vertex for E, backtrack	Delete E from queue F(3), G(4)
5	NO unvisited adjacent vertex for F, backtrack	Delete F from queue G(4)
6 7		Insert C, H into queue G(4), C(5), H(6) C(5), H(6)
  8		Insert D into queue C(5), H(6), D(7) H(6), D(7)
9	NO unvisited adjacent vertex for H, backtrack	Delete H from queue D(7)
	NO unvisited adjacent vertex for D, backtrack	Delete D from queue
	Queue becomes empty. Algorithm stops as all the nodes in the given graph are visited	

The BFS tree is as follows: (dotted lines are cross edges)



Applications of BFS:

- To check connectivity of a graph (number of times queue becomes empty tells the number of components in the graph)
- To check if a graph is acyclic. (no cross edges indicates no cycle)
- To find minimum edge path in a graph

Efficiency:

- Depends on the graph representation:
  - Array : $\Theta(n^2)$
  - List: $\Theta(n + e)$

Difference between DFS & BFS:

	DFS	BFS
Data structure	Stack	Queue
No. of vertex orderings	2 orderings	1 ordering
Edge types	Tree edge	Tree edge
	Back edge	Cross edge
Applications	Connectivity	Connectivity
	Acyclicity	Acyclicity
	Articulation points	Minimum edge paths
Efficiency for adjacency matrix	$\Theta(n^2)$	$\Theta(n^2)$
Efficiency for adjacency lists	$\Theta(n+e)$	$\Theta(n+e)$

5.4 Topological Sorting

Description:

Topological sorting is a sorting method to list the vertices of the graph 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.

NOTE: There is no solution for topological sorting if there is a cycle in the digraph . [MUST be a DAG]

Topological sorting problem can be solved by using

1. DFS method
2. Source removal method

DFS Method:

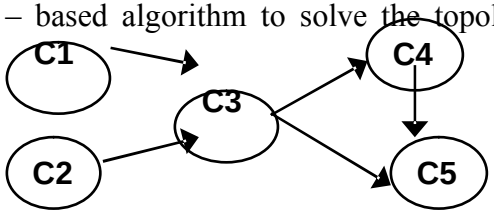
- Perform DFS traversal and note the order in which vertices become dead

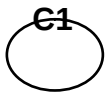
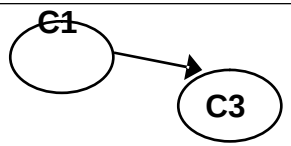
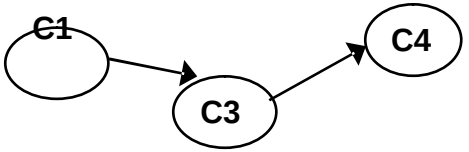
ends (popped order)

- Reverse the order, yield the topological sorting.

Example:

Apply DFS – based algorithm to solve the topological sorting problem for the given graph:

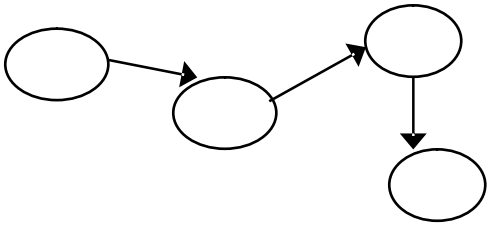


Step	Graph	Remarks
1		Insert C1 into stack C1(1)
2		Insert C2 into stack C2 (2) C1(1)
3		Insert C4 into stack C4 (3) C2 (2)
4		

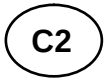
C1

C4

C3

		C1(1)
		Insert C5 into stack C5 (4) C4 (3) C2 (2) C1(1)

5	NO unvisited adjacent vertex for C5, backtrack	Delete C5 from stack C5 (4, 1) C4 (3) C2 (2) C1(1)
6	NO unvisited adjacent vertex for C4, backtrack	Delete C4 from stack C5 (4, 1) C4 (3, 2) C2 (2) C1(1)
7	NO unvisited adjacent vertex for C3, backtrack	Delete C3 from stack C5 (4, 1) C4 (3,2) C2 (2, 3) C1(1)
8	NO unvisited adjacent vertex for C1, backtrack	Delete C1 from stack

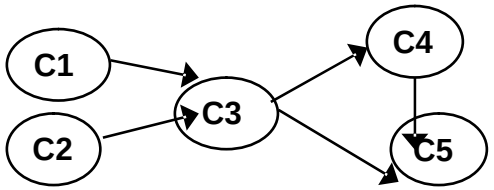
		C5 (4, 1) C4 (3,2) C2 (2, 3) C1(1, 4)
Stack becomes empty, but there is a node which is unvisited, therefore start the DFS again from arbitrarily selecting a unvisited node as source		
9		Insert C2 into stack C5 (4, 1) C4 (3,2) C2 (2, 3) C1(1, 4) C2(5)
10	NO unvisited adjacent vertex for C2, backtrack	Delete C2 from stack C5 (4, 1) C4 (3,2) C2 (2, 3) C1(1, 4) C2(5, 5)
Stack becomes empty, NO unvisited node left, therefore algorithm stops. The popping – off order is: C5, C4, C3, C1, C2, Topologically sorted list (reverse of pop order): C2, C1 C3 C4 C5		

Source removal method:

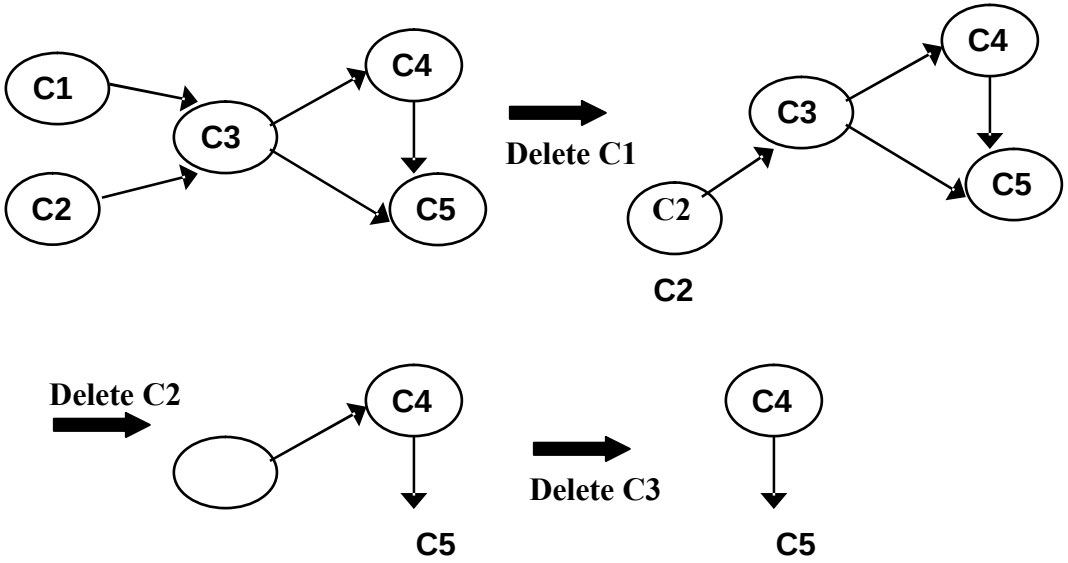
- Purely based on decrease & conquer
- Repeatedly identify in a remaining digraph a source, which is a vertex with no incoming edges
- Delete it along with all the edges outgoing from it.

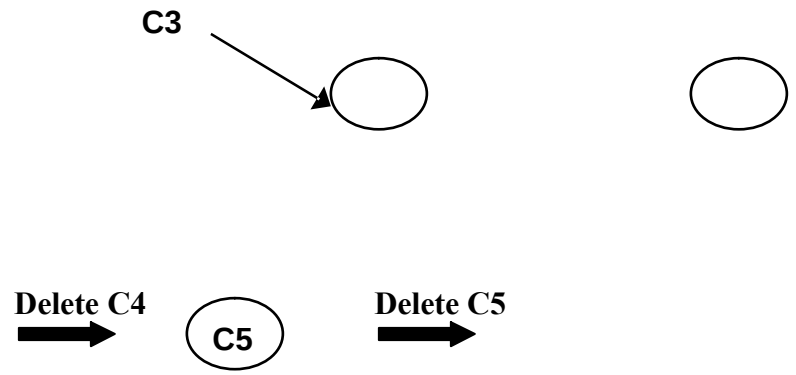
Example:

Apply Source removal – based algorithm to solve the topological sorting problem for the given graph:



Solution:





The topological order is C1, C2, C3, C4, C5

5.5 Space-Time Tradeoffs: Introduction

Two varieties of space-for-time algorithms:

- input enhancement — preprocess the input (or its part) to store some info to be used later in solving the problem
    - counting sorts
    - string searching algorithms
- pre structuring — preprocess the input to make accessing its elements easier

- hashing
- indexing schemes (e.g., B-trees)

5.6 SORTING BY COUNTING

Assume elements to be sorted belong to a known set of small values between *l* and *u*, with potential duplication Constraint: we cannot overwrite the original list Distribution Counting: compute the frequency of each element and later accumulate sum of frequencies (distribution)

Algorithm:

```
for j ← 0  to u-l  do D[j] ← 0                          // init frequencies
for i ← 0  to n-l do D[A[i]-l] ← D[A[i] - l] + 1 // compute frequencies
for j ← 1  to u-l  do D[j] ← D[j-1] + D[j]          // reuse for distribution
for i ← n-l downto 0  do
        j ← A[i] - l
        S[D[j] - 1] ← A[i]
        D[j] ← D[j] - 1
        return S
```

Example: A =

13	11	12	13	12	12

Array Values	11	12	13
Frequencies	1	3	2
Distribution	1	4	6

	D[0]	D[1]	D[2]	S[0]	S[1]	S[2]	S[3]	S[4]	S[5]
A[5] = 12	1	4	**6**				**12**		
A[4] = 12	1	3	**6**			**12**			
A[3] = 13	1	**2**	6						**13**
A[2] = 12	1	2	**5**		**12**				
A[1] = 11		**1**	**5**	**11**					
A[0] = 13	**0**	**1**	5					**13**	

Efficiency: $\Theta(n)$

- Best so far but only for specific types of input

5.7 INPUT ENHANCEMENT IN STRING MATCHING

Horspool's Algorithm

A simplified version of Boyer-Moore algorithm:

- **preprocesses pattern to generate a shift table that determines how much to shift the pattern when a mismatch occurs**
- **always makes a shift based on the text's character *c* aligned with the <u>last</u> compared (mismatched) character in the pattern according to the shift table's entry for *c***

How far to shift?

Look at first (rightmost) character in text that was compared:

- The character is not in the pattern

```
.....c.....................   (c not in pattern)
BAOBAB
```

- The character is in the pattern (but not the rightmost)

```
.....O.....................   (O occurs once in pattern)
BAOBAB
.....A.....................   (A occurs twice in pattern)
BAOBAB
```

- The rightmost characters do match

```
.....B.....................
BAOBAB
```

Shift table

- Shift sizes can be precomputed by the formula

$t(c) =$ { distance from c's rightmost occurrence in pattern among its first m-1 characters to its right end ; pattern's length m, otherwise

by scanning pattern before search begins and stored in a table called *shift table*. After the shift, the right end of pattern is $t(c)$ positions to the right of the last compared character in text.

- Shift table is indexed by text and pattern alphabet
Eg, for BAOBAB :

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6

Example of Horspool's algorithm

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	_
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6	6

```
BARD LOVED BANANAS
BAOBAB
      BAOBAB
           BAOBAB
             BAOBAB  (unsuccessful search)
```

If k characters are matched before the mismatch, then the shift distance is $d_1 = t(c) - k$.

```
.....czyx.....   k
...c....bzyx
    t(c)
...c....bzyx
```

Note that the shift could be negative!
E.g. if text = ...ABABAB...