Decrease & Conquer :-

General Idea: reduce the problem to smaller instance → solve smaller instance → extend sol^n to problem

a) Bottom-Up: Iterative
b) Top-Down: Recursive
c) Inductive / Incremental approach

Variations:
- Decrease by const
- Decrease by const. factor
- Decrease var. size

1) Insertion Sort: ⊛
   a) Idea / Example
   b) Algorithm
   c) Implementation (same as b)
   d) Analysis

2) Topographical Sort: (directed acyclic graphs)
   a) DFS-based
   b) Source Elimination

3) Algorithms for generating P and C:
   a) Johnson Trotter ⊛ } Permutation
   b) Lexographic permute ⊛ } Permutation
   c) Gray Code ⊛ — Subset gen.

4) Decrease by Const. Factor:
   a) Binary Search ⊛
   b) Fake coin Prob.
   c) Russian Peasant Multiplication
   d) Josephus Prob.

1)

a) → Compare current elem w/ largest val in sorted array
→ shift all elems > curr value by one position.

- Inplace
- Stable
- Best elementary sort

10 3 8 12 24 2
3 10 8 12 24 2
3 8 10 12 24 2
3 8 10 12 24 2
3 8 10 12 24 2
2 3 8 10 12 24

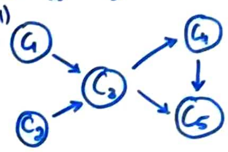
b)
```
for i ← 1 to n-1 do
    v ← A[i] ; j ← i-1
    while j ≥ 0 and A[j] > v do
        A[j+1] ← A[j]   (shifts by 1 pos)
        j ← j-1
    A[j+1] ← v
```

d) $\Theta(n^2)$: worst $(n(n-1)/2)$ / avg. $(n^2/4)$; $\Theta(n)$: best $(n-1)$

2) ⊛ Implementation not covered here.

a)
- Same efficiency as DFS

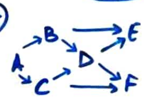
Ex: (1)



→ perform DFS
→ order in which they become dead-ends (popped out)
→ reverse order = solution

Push: C1 C2; C3; C4; C5
pop: C5 C4 C3 C1 C2
Sol^n: C2 C1 → C3 → C4 → C5

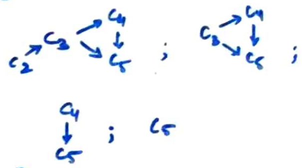
(2)



Push: A B D E F C
Pop : E F D C B A
Sol^n : A → B → C → D → F → E

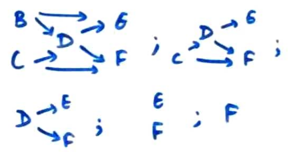
b)
- Identify source (Indegree = 0) delete source vertex and outgoing edges.
- removal order = solution.

Ex(1):
→ Same.



Sol^n: C1 → C2 → C3 → C4 → C5

(2): ⊛ multiple sol^ns possible



Sol^n: A → B → C → D → E → F

Decrease & Conquer :-

General Idea: reduce the problem to smaller instance → solve smaller instance → extend sol^n to problem

a) Bottom-Up: Iterative
b) Top-Down: Recursive
c) Inductive / Incremental approach

Variations

- Decrease by Const
- Decrease by const. factor
- Decrease var. size

1) Insertion Sort:
   a) Idea / Example
   b) Algorithm
   c) Implementation (same as b)
   d) Analysis

2) Topographical Sort: (directed acyclic graphs)
   a) DFS-based
   b) Source Elimination

3) Algorithms for generating P and C:
   a) Johnson Trotter
   b) Lexographic permute
   c) Gray Code — Subset gen.
   d) Minimal change method
   e) Heap Permute

4) Decrease by Const. Factor:
   a) Binary Search
   b) Fake coin Prob.
   c) Russian Peasant Multiplica^n
   d) Josephus Prob.

3)

d) Ex: {A, B, C} $\theta(n!)$: efficiency

· insert from R → L

ABC, ACB, CAB

insert from L → R

CBA, BCA, BAC

a) · elements w/ arrows (initially ←)

Ex: (1, 2, 3) : ideally (1,2,3,4) is better to understand this completely

$\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}$
$\overleftarrow{1}\ \overleftarrow{3}\ \overleftarrow{2}$
$\overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{2}$
$\overrightarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}$
$\overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{1}$
$\overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{3}$

- always choose highest possible no. first.
- Aft. swap, change dire^n of arrow of ALL elements greater than swapped elem.
- elem can only move in dire^n of arrow & iff elem. is smaller.

Algo: Johnson Trotter (n) → See TBK pg. 145 (pdf pg. 169)

.... a_{n-3} a_{n-2} a_{n-1} a_n (initial config.)

b) • permutations generated in order.

• $a_{n-1} < a_n$ → swap

$a_{n-1} > a_n$ → find a_{n-2} and replace by imm. greater value from a_{n-1} or a_n and arrange a_{n-1} & a_n in ascending order.

• stop on reaching a descending order.

Space & Time Tradeoffs :-

I) Input Enhancement

1) Comparison Counting Sorting :-

- Count no. of elements smaller than the element in array for each element ; • Complexity : $O(n^2)$

Ex:

array :	60	20	41	19	1	31	
Count :	5	2	4	1	0	3	→ extra Space
Sorted :	1	19	20	31	41	60	
	(0)	(1)	(2)	(3)	(4)	(5)	

Algorithm: I/P (A, n)

```
count [n] = {0}  //initialise all values to 0
S[n] = {0}  // sorted array
for i ← 0 to n-2
    for j ← i+1 to n-1
        if  A[i] > A[j]  //search for smaller elements
              count[i]++
        else
              count[j]++  // else increase count
                             of other element
for i ← 0 to n-1
    S[count[i]] = A[i]  // places elements at
                           corresponding indices
                           using count array
```

2) Distribution Counting Sorting :-

- Make a frequency distribu^n table

Ex: 11 12 13 12 11 12 12 13

Symbol	11	12	13
freq.	2	4	2
dist. val	2	6	8 → D[ ]

~~11~~	~~12~~	~~13~~	~~12~~	~~11~~	~~12~~	~~12~~	~~13~~
1	5	7	4	0	3	2	6

pos. :-	0	1	2	3	4	5	6	7
	-	11	-	-	-	-	-	-
	-	11	-	-	-	12	-	-
	-	11	-	-	-	12	-	13
	-	11	-	-	12	12	-	13
	11	11	-	-	12	12	-	13
	11	11	-	12	12	12	-	13
	11	11	12	12	12	12	-	13
Sorted →	11	11	12	12	12	12	13	13

Algo :- I/P (A, u, l, n)

```
D[u-l+1] = {0}  //init freq = 0
for i ← 0 to n-1 do
    D[A[i]-l] ← D[A[i]-l]+1
                  // compute freq.
for j ← 1 to u-l do
    D[j] ← D[j-1] + D[j]  // cf.
for i ← n-1 down to 0 do
    j ← A[i]-l  // pos. of A[i] in D[ ]
```

$S[D[j]-1] \leftarrow A[i]$ // store elem in correct posi^n using D[].

$D[j]$ -- // reduce cf.

// ← end of for loop → //

return S

- Complexity : $O(n)$