Decrease-and-Conquer

1. Reduce problem instance to smaller instance of the same problem
2. Solve smaller instance
3. Extend solution of smaller instance to obtain solution to original instance

- Can be implemented either top-down or bottom-up
- Also referred to as *inductive* or *incremental* approach
3 Types of Decrease and Conquer

- **Decrease by a constant** (usually by 1):
  - insertion sort
  - topological sorting
  - algorithms for generating permutations, subsets

- **Decrease by a constant factor** (usually by half)
  - binary search and bisection method
  - exponentiation by squaring
  - multiplication à la russe

- **Variable-size decrease**
  - Euclid’s algorithm
  - selection by partition
  - Nim-like games
What’s the difference?

Consider the problem of exponentiation: Compute $a^n$

- **Brute Force:**
- **Divide and conquer:**
  - Decrease by one:
  - Decrease by constant factor:
Insertion Sort
To sort array $A[0..n-1]$, sort $A[0..n-2]$ recursively and then insert $A[n-1]$ in its proper place among the sorted $A[0..n-2]$

- Usually implemented bottom up (nonrecursively)

Example: Sort 6, 4, 1, 8, 5

```
6 | 4 1 8 5
4 6 | 1 8 5
1 4 6 | 8 5
1 4 6 8 | 5
1 4 5 6 8
```
Pseudocode of Insertion Sort

**ALGORITHM**  \textit{InsertionSort}(A[0..n - 1])

// Sorts a given array by insertion sort
// Input: An array A[0..n - 1] of n orderable elements
// Output: Array A[0..n - 1] sorted in nondecreasing order

\begin{algorithmic}
  \FOR{$i \leftarrow 1$ \textbf{to} $n - 1$}
    \STATE $v \leftarrow A[i]$
    \STATE $j \leftarrow i - 1$
    \WHILE{$j \geq 0$ \textbf{and} $A[j] > v$}
      \STATE $A[j + 1] \leftarrow A[j]$
      \STATE $j \leftarrow j - 1$
    \ENDWHILE
    \STATE $A[j + 1] \leftarrow v$
  \ENDFOR
\end{algorithmic}
Analysis of Insertion Sort

- Time efficiency
  
  \[ C_{\text{worst}}(n) = \frac{n(n-1)}{2} \in \Theta(n^2) \]
  
  \[ C_{\text{avg}}(n) \approx \frac{n^2}{4} \in \Theta(n^2) \]
  
  \[ C_{\text{best}}(n) = n - 1 \in \Theta(n) \quad \text{(also fast on almost sorted arrays)} \]

- Space efficiency: in-place

- Stability: yes

- Best elementary sorting algorithm overall

- Binary insertion sort

Dags and Topological Sorting

A dag: a directed acyclic graph, i.e. a directed graph with no (directed) cycles

Arise in modeling many problems that involve prerequisite constraints (construction projects, document version control)

Vertices of a dag can be linearly ordered so that for every edge its starting vertex is listed before its ending vertex (topological sorting). Being a dag is also a necessary condition for topological sorting be possible.
Topological Sorting Example
Order the following items in a food chain

plankton -> shrimp -> fish -> human -> sheep -> wheat -> tiger
DFS-based Algorithm

DFS-based algorithm for topological sorting

- Perform DFS traversal, noting the order vertices are popped off the traversal stack
- Reverse order solves topological sorting problem
- Back edges encountered? → NOT a dag!

Example:

```
  a <-> b <-> c <-> d
  |     |     |     |
  e <-> f <-> g <-> h
```

Efficiency:
Source Removal Algorithm

Source removal algorithm

Repeatedly identify and remove a source (a vertex with no incoming edges) and all the edges incident to it until either no vertex is left (problem is solved) or there is no source among remaining vertices (not a dag)

Example:

Efficiency: same as efficiency of the DFS-based algorithm
Generating Permutations

**Minimal-change decrease-by-one algorithm**

If $n = 1$ return 1; otherwise, generate recursively the list of all permutations of 12...n-1 and then insert $n$ into each of those permutations by starting with inserting $n$ into 12...n-1 by moving right to left and then switching direction for each new permutation.

Example: $n=3$

start

insert 2 into 1 right to left

insert 3 into 12 right to left

insert 3 into 21 left to right

1

12 21

123 132 312

321 231 213
Other permutation generating algorithms

- Johnson-Trotter (p. 145)
- Lexicographic-order algorithm (p. 146)
- Heap’s algorithm (Problem 4 in Exercises 4.3)
Generating Subsets

**Binary reflected Gray code**: minimal-change algorithm for generating $2^n$ bit strings corresponding to all the subsets of an $n$-element set where $n > 0$

If $n=1$ make list $L$ of two bit strings 0 and 1

else

  generate recursively list $L_1$ of bit strings of length $n-1$
  copy list $L_1$ in reverse order to get list $L_2$
  add 0 in front of each bit string in list $L_1$
  add 1 in front of each bit string in list $L_2$
  append $L_2$ to $L_1$ to get $L$

return $L$