REDUCE-AND-CONQUER
Class of related strategies in which a solution to a problem is designed from solutions of more manageable components of the problem

Two strategies:
• Decrease–and-conquer (ch 4): find solution to small instance of problem and build general solution from it.
• Divide-and-conquer (ch 5): divide problem into subproblems, solve them, and combine subsolutions into general solution.

DECREASE-AND-CONQUER

Approach
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

Notes:
• Can be implemented either top-down (recursively) or bottom-up (iteratively)
• Also referred to as inductive or incremental approach

INSERTION SORT
Explanation: 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]

Top-down:
Bottom-up:

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

// Sorts a given array by insertion sort
// Input: An array A[0..n-1] of \textit{n} orderable elements
// Output: Array A[0..n-1] sorted in nondecreasing order
for \( i \leftarrow 1 \) to \( n-1 \) do
  \( v \leftarrow A[i] \)
  \( j \leftarrow i - 1 \)
  \textbf{while} \( j \geq 0 \) \textbf{and} \( A[j] > v \) \textbf{do}
    \( A[j+1] \leftarrow A[j] \)
    \( j \leftarrow j - 1 \)
  \( A[j+1] \leftarrow v \)

**Efficiency**

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

- Space efficiency: in-place
- Stability: yes
- Best elementary sorting algorithm overall

Note: this was a decrease by 1 algorithm

**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
COMPARE APPROACHES

Calculate $a^n$
- Brute Force:
  - Decrease by one:
  - Divide and conquer:
  - Decrease by constant factor:

DAGS AND TOPOLOGICAL SORTING

Definition of Dag

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

- A topological sort of a graph is a linear ordering of the vertices so that for every edge its starting vertex is listed before its ending vertex in the ordering.
- **Theorem**: a graph can be topologically sorted iff it is a dag.
- Example: order the following items in a food chain

```
  tiger
    ▼
  human
    ▼
  fish
    ▼
shrimp
    ▼
plankton
    ▼
  wheat
  ▼
sheep
```

Source removal algorithm (decrease by one-and-conquer)

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)

Efficiencies

- Size: \( V = \# \) of vertices and \( E = \# \) of edges
- Efficiency of DFS algorithm = efficiency of DFS traversal:
  - adjacency matrices: \( \Theta(V^2) \)
  - adjacency lists: \( \Theta(|V|+|E|) \)
- Efficiency of source removal 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

```
| h | a |
| g | f |
```

```
| b | c |
| e | d |
```

- What if diagram is not a DAG?

```
| h | a |
| g | f |
```

```
| b | c |
| e | d |
```

GENERATING PERMUTATIONS

Minimal-change decrease-by-one Johnson-Trotter algorithm:

If \( n = 1 \) return 1;
otherwise,
- generate the string 12\ldots(n-1)
- then insert \( n \) into 12\ldots(n-1) starting by moving right to left and then
  switching direction when \( n \) reaches an edge.
- At each switch: before moving swap the two elements next to \( n \).

Example: \( n=4 \)

\[
\begin{array}{c|c}
\text{start} & 1 \\
\text{insert 2 into 1 right to left} & 12 \ 21 \\
\text{insert 3 into 12 right to left} & 123 \ 132 \ 312 \leftarrow \text{swap} \\
& 321 \ 231 \ 213 \\
\text{insert 4 into 123 right to left} & 1234 \ 1243 \ 1423 \ 4123 \ 4213 \ 2413 \ 2143 \ 2134 \\
& 2314 \ 2341 \ 2431 \ 4231 \ 4321 \ 3421 \ 3241 \ 3214 \\
& 3124 \ 3142 \ 3412 \ 4312 \ 4132 \ 1432 \ 1342 \ 1324 \\
\end{array}
\]

Gray codes
- A “Gray code” for a set of combinatorial objects is an ordering of the objects
  in which each two consecutive objects differ in the minimal possible way.
- The Johnson-Trotter algorithm generates a Gray code for the list of all
  possible permutations of 1..n

Other Algorithms to generate Permutations
- Improvement of Johnson-Trotter by Even (Text p. 145) \( \leftarrow \) very efficient
- Lexicographic-order algorithm (Text p. 146)
- Heap’s algorithm (Problem 4 in Exercises 4.3)
- All are \( \Theta(n!) \) because this is the size of the problem
GENERATING SUBSETS

Knapsack problem: generate all subsets of \{1,\ldots,n\}

- Size of problem = number of subsets = 2^{n-1} (If \emptyset is excluded)

Decrease by 1-and-conquer:

- Generate all possible subsets of \{1, \ldots, n-1\}
- For each such subset, create an additional subset by adding n to it.

Bit string approach: (not decrease by 1-and-conquer)

- Represent each subset by a bit string of length n
- Start with 0 and add 1 until number 2^{n-1} is reached

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

Example for n=3

- generate recursively list L_1 of bit strings of length n-1=2:
  L_1 = (00, 01, 11, 10)
- copy list L_1 in reverse order to get list L_2
  L_2 = (10, 11, 01, 00)
- add 0 in front of each bit string in list L_1
  L_1 = (000, 001, 011, 010)
- add 1 in front of each bit string in list L_2
  L_2 = (110, 111, 101, 100)
- append L_2 to L_1 to get L
  L = (000, 001, 011, 010, 110, 111, 101, 100)
- return L
DECREASE BY A CONSTANT FACTOR

Instance size is reduced by the same factor (usually by half)

Binary Search

- Algorithm
  - Very efficient algorithm for searching for K in sorted array:
  - If \( K = A[n/2] \), stop (successful search);
    otherwise, continue searching by the same method
    in \( A[1..n/2-1] \) if \( K < A[n/2] \), or
    in \( A[n/2+1..n] \) if \( K > A[n/2] \)

- Time efficiency
  - worst-case recurrence: \( C_w(n) = 1 + C_w(\lceil n/2 \rceil) \), \( C_w(1) = 1 \)
    solution: \( C_w(n) = \lceil \log_2(n+1) \rceil \)
    This is VERY fast: e.g., \( C_w(10^6) = 20 \)
  - Optimal for searching a sorted array
  - Limitations: must be a sorted array (not linked list)

Russian Peasant Multiplication

- The problem: compute the product of two positive integers
- Can be solved by a decrease-by-half algorithm based on the following formulas.
  - For \( n=0 \): \( n \times m = 0 \)
  - For \( n=1 \): \( n \times m = m \)
  - For even values of \( n \): \( n \times m = \frac{n}{2} \times 2m \)
  - For odd values of \( n \): \( n \times m = \frac{n-1}{2} \times 2m + m \)

- Example: Compute \( 50 \times 65 \)

\[
\begin{array}{cc}
n & m \\
50 & 65 \\
25 & 130 \\
12 & 260 (+130) \\
6 & 520 \\
3 & 1040 \\
1 & 2080 (+1040) \\
\end{array}
\]

\[ = 3250 \]
VARIABLE-SIZE-DECREASE ALGORITHMS

Selection Problem

- Problem: Find the $k$-th smallest element in a list of $n$ numbers
  - Minimum: $k = 1$
  - Maximum: $k = n$
  - Median: $k = \lceil n/2 \rceil$

Example: 4, 1, 10, 8, 7, 12, 9, 2, 15  median =

The median is used in statistics as a measure of an average value of a sample. In fact, it is a better (more robust) indicator than the mean, which is used for the same purpose.

- Sorting-based algorithm:
  - Sort and return the $k$-th element
    Efficiency: (if sorted by mergesort): $\Theta(n \log n)$

- Quickselect: faster algorithm involving array partitioning

| all are $\leq A[s]$ | $A[s]$ | all are $\geq A[s]$ |

Assuming that the array is indexed from 0 to $n-1$ and $s$ is a split position obtained by the array partitioning:

If $s = k-1$, the problem is solved;  
if $s > k-1$, look for the $k$-th smallest element in the left part;  
if $s < k-1$, look for the $k-s$-th smallest element in the right part.  
Note: The algorithm can simply continue until $s = k-1$.

- There are two principal ways to partition an array:
  - One-directional scan (Lomuto’s partitioning algorithm)  
  - Two-directional scan (Hoare’s partitioning algorithm)
• Hoare’s partitioning algorithm:
  – Use the first element of the array as the pivot \((s = A[1])\)
  – Move simultaneously from the left and the right until you find
    o One element on the left which is bigger than \(s\)
    o One element on the right which is smaller than \(s\)
  – Swap them and continue until the left and the right meet
  – Swap \(A[1]\) with the largest element on the left

• Tracing Quickselect

  Find the median of \(4, 1, 10, 8, 7, 12, 9, 2, 15\)
  Here: \(n = 9, k = \lceil 9/2 \rceil = 5, k - 1 = 4\)

  \[
  \begin{array}{cccccccc}
  0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
  \hline
  4 & 1 & 10 & 8 & 7 & 12 & 9 & 2 & 15 \\
  4 & 1 & 2 & 8 & 7 & 12 & 9 & 10 & 15 \\
  2 & 1 & 4 & 8 & 7 & 12 & 9 & 10 & 15 \\
  2 & 1 & 4 & 8 & 7 & 12 & 9 & 10 & 15 \\
  \end{array}
  \]

• Efficiency of Quickselect
  – Best case (1st element is the solution)
    \[C(n) = n + 1\] \(C(n) \in \Theta(n)\)
  – Average case (average split in the middle):
    \[C(n) = C(n/2) + (n + 1)\] \(C(n) \in \Theta(n)\)
  – Worst case (degenerate split – sorted or inversely sorted):
    \[C(n) \in \Theta(n^2)\]
  – A more sophisticated choice of the pivot leads to a complicated algorithm with \(\Theta(n)\) worst-case efficiency.
Interpolation Search

- Searches a sorted array similar to binary search but estimates location of the search key in \( A[l..r] \) by using its value \( v \). Specifically, the values of the array’s elements are assumed to grow linearly from \( A[l] \) to \( A[r] \) and the location of \( v \) is estimated as the x-coordinate of the point on the straight line through \((l, A[l])\) and \((r, A[r])\) whose y-coordinate is \( v \):

\[
x = l + \left\lfloor \frac{(v - A[l])(r - l)}{(A[r] - A[l])} \right\rfloor
\]

*Figure 4.14* Index computation in interpolation search.

- Analysis of Interpolation Search
  - Efficiency
    - best case (values of array elements do grow linearly): \( C(n) = 1 \)
    - average case: \( C(n) < \log_2 \log_2 n + 1 \)
    - worst case: \( C(n) = n \)
  - Preferable to binary search only for VERY large arrays and/or expensive comparisons