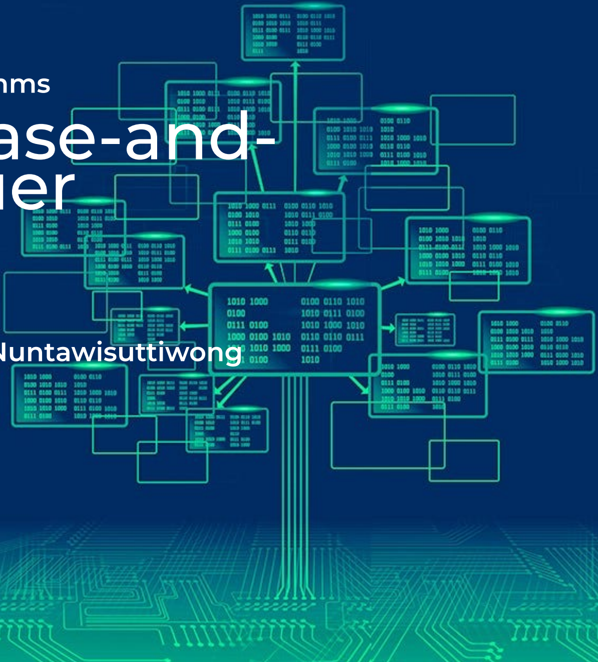
CPE 231 Algorithms

Decrease-and-Conquer



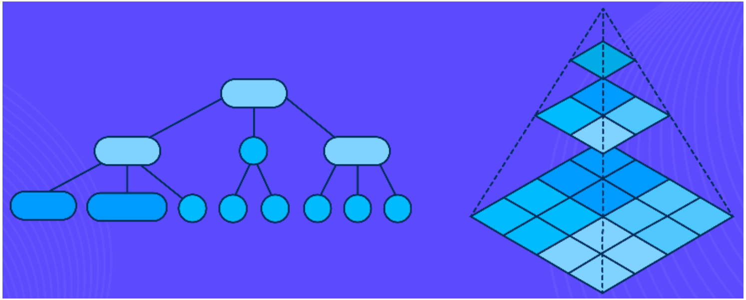
Dr. Taweechai Nuntawisuttiwong

Contents

Decrease-and-Conquer

Definition

The decrease-and-conquer technique is a problem-solving strategy that simplifies a problem by reducing it to a smaller instance of the same problem, then solving the smaller instance and using that solution to address the original problem.



Two Main Implementation Approaches

Top-Down Approach (Recursive):

- The problem is repeatedly broken down until the base case is reached (smallest version of the problem).
- The smaller solutions are combined to form the solution to the larger problem.

Bottom-Up Approach (Iterative):

- Start with solving the smallest possible subproblem and build the solution iteratively.
- More efficient in practice for many problems compared to recursion.

Variations of Decrease-and-Conquer

Decrease by a Constant:

- Reduce the problem size by a constant amount (typically 1).
- Example: Insertion sort.

Decrease by a Constant Factor:

- Reduce the problem size by a constant factor (typically by half).
- Example: Binary search.

Variable Size Decrease:

- The amount of decrease changes at each step.
- Example: Euclid’s algorithm for GCD.

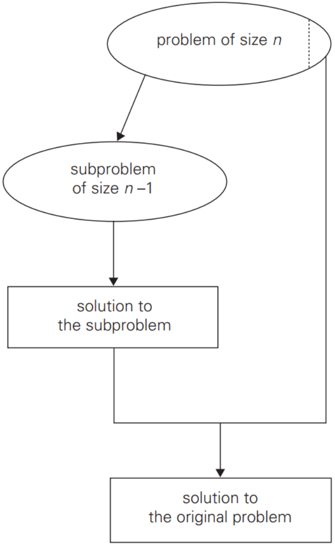
Decrease by a Constant

In this method, the problem is reduced by one unit in each step.

Example: Exponentiation

Compute a^n by reducing the exponent by 1 each time.

$$f(n) = \begin{cases} f(n-1) \cdot a & , n > 0 \\ 1 & , n = 0 \end{cases}$$



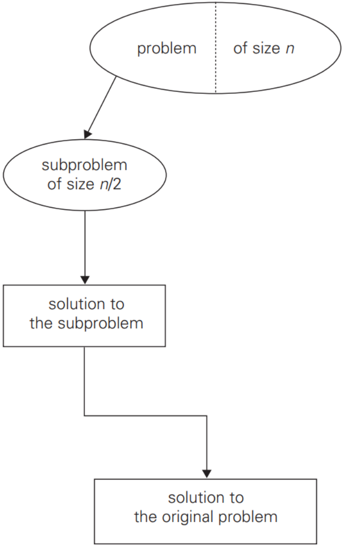
Decrease by a Constant Factor

The problem size is reduced by a constant factor in each iteration.

Example: Exponentiation by Squaring

Compute a^n by reducing the size by half.

$$a^n = \begin{cases} (a^{n/2})^2 & , n \text{ is even} \\ (a^{(n-1)/2})^2 \cdot a & , n \text{ is odd} \\ 1 & , n = 0 \end{cases}$$



Variable Size Decrease

The size of the problem is reduced by varying amounts in each step.

Example: Euclid's Algorithm

- Compute GCD using the formula:

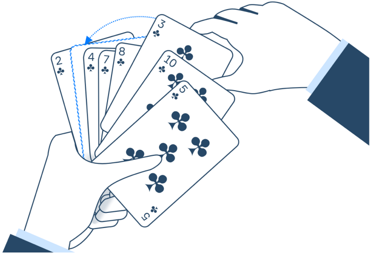
$$gcd(m, n) = gcd(n, m \mod n)$$

- The size of the input reduces unpredictably with each iteration.

Insertion Sort

Insertion Sort

- Insertion sort is a simple comparison-based sorting algorithm that builds the sorted array one element at a time by inserting each element into its correct position.
- At each step, the current element is compared to its predecessors and placed in its correct position relative to the sorted portion of the array.
- The **decrease-by-one** strategy, where we solve a smaller instance of the problem and extend it to solve the larger problem.



How Insertion Sort Works

1. **Assumption**: The array is partially sorted up to element $A[i-1]$.
2. **Next Step**: Insert the current element $A[i]$ into its correct position within the sorted subarray.
3. **Process**: Compare $A[i]$ with the elements to its left (from right to left) and shift all larger elements one position to the right until $A[i]$ finds its correct place.

$$\underbrace{A[0] \le \cdots \le A[j]}_{\text{smaller than or equal to } A[i]} < \underbrace{A[j+1] \le \cdots \le A[i-1]}_{\text{greater than } A[i]} \mid A[i] \cdots A[n-1]$$

Pseudocode of Insertion Sort

ALGORITHM *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
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**
    $A[j+1] \leftarrow A[j]$
    $j \leftarrow j-1$
  $A[j+1] \leftarrow v$

Example

Initial Array: [89, 45, 68, 29, 34, 17]

Time Complexity of Insertion Sort

Best Case:

Occurs when the array is already sorted. Each element is compared once.

Worst Case:

Occurs when the array is sorted in reverse order. Every element must be compared with all its

Average Case:

For a randomly ordered array, insertion sort performs about half as many comparisons as in the worst case.

Strengths and Weaknesses

Strengths:

- Simple and easy to implement.
- Performs well for small or nearly sorted datasets.
- Efficient for small datasets or as the final step of more advanced sorting algorithms.

Weaknesses:

- Poor performance on large datasets due to its quadratic time complexity ($O(n^2)$).
- Not suitable for very large arrays compared to more advanced algorithms like merge sort or quicksort.

Algorithms for Generating Combinatorial Objects

Combinatorial Objects

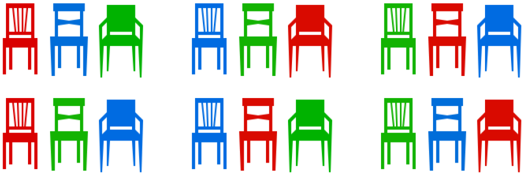
- Combinatorial objects include permutations, combinations, and subsets that are essential in problems involving choices or arrangements.
- Widely used in problems like exhaustive search, optimization, and mathematical combinatorics.
- Learn algorithms to generate these objects efficiently, understanding their computational challenges.

Permutations of a Set

A permutation is an arrangement of the elements of a set in a specific order. Given a set of size n, a permutation is any possible reordering of the elements.

Example: Consider the set $\{1, 2, 3\}$. Its permutations are

$$123,\ 132,\ 213,\ 231,\ 312,\ 321$$



Generating Permutations

- Generate permutations of $n-1$ elements and insert the n-th element into every possible position.
- This ensures all permutations are unique and covers all $n!$ possibilities.

start	1		
insert 2 into 1 right to left	12	21	
insert 3 into 12 right to left	123	132	312
insert 3 into 21 left to right	321	231	213

Johnson-Trotter Algorithm

- The Johnson-Trotter algorithm is an efficient method for generating all permutations of a set of n elements by making minimal changes between successive permutations.
- Each permutation generated differs from the previous one by swapping only two adjacent elements, making it a **minimal-change** algorithm.
- The minimal-change approach optimizes the algorithm for scenarios where updating permutations incrementally is essential, such as in exhaustive search problems (e.g., the Traveling Salesman Problem).

Johnson-Trotter Algorithm

ALGORITHM *JohnsonTrotter*(n)

//Implements Johnson-Trotter algorithm for generating permutations
//Input: A positive integer n
//Output: A list of all permutations of $\{1, \ldots, n\}$
initialize the first permutation with $\overleftarrow{1}\,\overleftarrow{2} \ldots \overleftarrow{n}$
while the last permutation has a mobile element **do**
    find its largest mobile element k
    swap k with the adjacent element k's arrow points to
    reverse the direction of all the elements that are larger than k
    add the new permutation to the list

Johnson-Trotter Algorithm

Mobile Elements:

- Each element in the permutation has a direction (left or right) indicated by an arrow.
- An element is called mobile if its arrow points to a smaller adjacent number.
- Largest Mobile Element: The largest number in the permutation that is mobile is swapped with the adjacent element in the direction of its arrow.

$$\overrightarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}\ \overleftarrow{1}$$

Johnson-Trotter Algorithm

Swapping Elements:

- The largest mobile element is swapped with the adjacent element in the direction of its arrow.
- After swapping, the directions of all elements larger than the swapped element are reversed.

Termination Condition:

The algorithm terminates when there are no more mobile elements, meaning that all permutations have been generated.

Example:

$$\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{\mathbf{3}}\quad \overleftarrow{1}\,\overleftarrow{\mathbf{3}}\,\overleftarrow{2}\quad \overleftarrow{3}\,\overleftarrow{1}\,\overleftarrow{\mathbf{2}}\quad \overrightarrow{\mathbf{3}}\,\overleftarrow{2}\,\overleftarrow{1}\quad \overleftarrow{2}\,\overrightarrow{\mathbf{3}}\,\overleftarrow{1}\quad \overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{3}.$$

Lexicographic Permutations

Lexicographic permutations are ordered permutations of a set where the elements are arranged as they would appear in dictionary (or alphabetical) order. This order is determined by comparing the elements as if they were characters in a string.

Example: For a set $\{1, 2, 3\}$, the lexicographic order of permutations is:

$$123,\ 132,\ 213,\ 231,\ 312,\ 321$$

- This is the order you would see if the numbers were "sorted" in increasing order like words in a dictionary.

Lexicographic Permutations

ALGORITHM *LexicographicPermute*(n)

//Generates permutations in lexicographic order
//Input: A positive integer n
//Output: A list of all permutations of $\{1, \ldots, n\}$ in lexicographic order
initialize the first permutation with $12 \ldots n$
while last permutation has two consecutive elements in increasing order **do**
  let i be its largest index such that $a_i < a_{i+1}$ //$a_{i+1} > a_{i+2} > \cdots > a_n$
  find the largest index j such that $a_i < a_j$ //$j \geq i + 1$ since $a_i < a_{i+1}$
  swap a_i with a_j //$a_{i+1}a_{i+2} \ldots a_n$ will remain in decreasing order
  reverse the order of the elements from a_{i+1} to a_n inclusive
  add the new permutation to the list

Lexicographic Permutations

Advantages

1. **Natural Order**: The permutations are generated in an order that makes sense when compared to a dictionary.
2. **Efficient Successor Generation**: Finding the next permutation in lexicographic order requires only local modifications (finding and swapping two elements, followed by reversing a suffix), making it efficient to implement.
3. **Useful in Combinatorial Problems**: Lexicographic ordering is especially useful in problems where an ordered exploration of permutations is required.

Decrease-by-a-Constant-Factor Algorithms

Decrease-by-a-Constant-Factor Algorithms

- These algorithms reduce the problem size by a constant factor in each step, typically halving the problem size.
- Typically run in logarithmic time $O(\log n)$, which makes them highly efficient.
- **Examples**: Binary Search, Russian Peasant Multiplication, Exponentiation by Squaring.

Binary Search

- **Task**: Search for a key K in a sorted array of size n.
- **Strategy**: Divide the array into two halves and compare the middle element with K. Discard one half depending on the result and continue the search in the remaining half.

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

Binary Search

ALGORITHM $BinarySearch(A[0..n-1], K)$

```
//Implements nonrecursive binary search
//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 ← 0;   r ← n − 1
while l ≤ r do
    m ← ⌊(l + r)/2⌋
    if K = A[m] return m
    else if K < A[m] r ← m − 1
    else l ← m + 1
return −1
```

Binary Search

Example: searching for K = 70 in the array

$$[3, 14, 27, 31, 39, 42, 55, 70, 74, 81, 85, 93, 98]$$

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
iteration 1													
iteration 2													
iteration 3													

Binary Search

Time Complexity

- **Worst-Case Complexity**: $O(\log n)$ because the array is halved at each step.
- **Best-Case Complexity**: $O(1)$ if the key is found at the middle element in the first comparison.
- **Average-Case Complexity**: Also $O(\log n)$.

Russian Peasant Multiplication

- **Task**: Multiply two positive integers n and m using an unconventional, iterative approach based on halving and doubling.
- **Key Operations**:
  - **Halving** n and **doubling** m at each step.

$$n \cdot m = \frac{n}{2} \cdot 2m$$

  - Add the doubled values of m when n is odd.

$$n \cdot m = \frac{n-1}{2} \cdot 2m + m$$

Russian Peasant Multiplication

Example: Compute $50 \cdot 65$

n	m
50	65

Variable-Size-Decrease Algorithms

Variable-Size-Decrease Algorithms

- A class of algorithms where the reduction in problem size varies from one iteration to another, rather than being fixed (like halving or reducing by a constant factor).
- **Examples**:
  - Euclid’s Algorithm for the Greatest Common Divisor (GCD).
  - Selection algorithms like Quickselect.

Computing a Median and the Selection Problem

- **Problem**: Given a list of n numbers, find the k-th smallest element, known as the k-th order statistic.
- **Special Case**: When $k = n/2$, the problem becomes finding the median, the element that separates the lower half from the upper half of the dataset.

Efficient Approach Using Partitioning

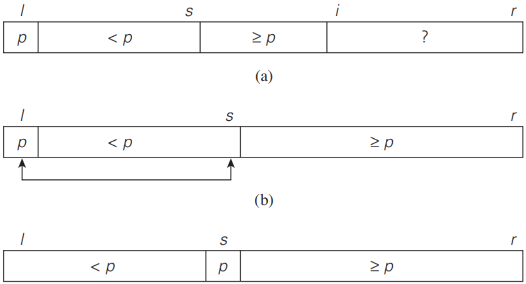
- Instead of sorting the entire list, partition it around a pivot, similar to the approach used in Quicksort.
- **Partitioning**: Rearrange the list such that:
  - Elements smaller than or equal to the pivot are on the left.
  - Elements larger than or equal to the pivot are on the right.
- Use the partitioning result to focus only on the part of the list containing the k-th smallest element.

Lomuto Partitioning Algorithm

A partitioning algorithm that divides the array into two parts based on a pivot element.

Steps:

1. Select the pivot (first element of the subarray).
2. Initialize two segments:
   - Elements **less than** the pivot.
   - Elements **greater than or equal to** the pivot.
3. Scan the array and place elements in their correct segment by swapping.
4. Finally, place the pivot in its correct position.



Lomuto Partitioning Algorithm

ALGORITHM *LomutoPartition*$(A[l..r])$

//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray $A[l..r]$ of array $A[0..n-1]$, defined by its left and right
//   indices l and r $(l \le r)$
//Output: Partition of $A[l..r]$ and the new position of the pivot
$p \leftarrow A[l]$
$s \leftarrow l$
for $i \leftarrow l+1$ **to** r **do**
    if $A[i] < p$
        $s \leftarrow s+1$;   swap$(A[s], A[i])$
swap$(A[l], A[s])$
return s

Using Partitioning to Solve the Selection Problem

Once the list is partitioned, use the position of the pivot to determine if the k-th smallest element lies to its left or right.

ALGORITHM $\quad Quickselect(A[l..r], k)$

//Solves the selection problem by recursive partition-based algorithm
//Input: Subarray $A[l..r]$ of array $A[0..n-1]$ of orderable elements and
// integer k $(1 \le k \le r - l + 1)$
//Output: The value of the kth smallest element in $A[l..r]$
$s \leftarrow LomutoPartition(A[l..r])$ //or another partition algorithm
if $s = k - 1$ **return** $A[s]$
else if $s > l + k - 1$ $Quickselect(A[l..s-1], k)$
else $Quickselect(A[s+1..r], k-1-s)$

Finding the Median Using Quickselect

Example: Apply the partition-based algorithm to find the median of the following list of nine numbers: 4, 1, 10, 8, 7, 12, 9, 2, 15.

0	1	2	3	4	5	6	7	8

Finding the Median Using Quickselect

Example: Apply the partition-based algorithm to find the median of the following list of nine numbers: 4, 1, 10, 8, 7, 12, 9, 2, 15.

0	1	2	3	4	5	6	7	8

Finding the Median Using Quickselect

Time Complexity of Quickselect

- **Best Case**: $O(n)$ – A good partitioning results in solving the problem in linear time.
- **Worst Case**: $O(n^2)$ – Poor partitioning leads to highly unbalanced splits.
- **Average Case**: $O(n)$ – With good pivot selection, most real-world cases have linear time complexity.

Finding the Median Using Quickselect

Applications of the Selection Problem

- **Statistical Analysis**: Computing medians and other order statistics is crucial in fields like data analysis and finance.
- **K-th Smallest Element in Arrays**: Useful in algorithms that need to identify extreme values without sorting (e.g., nearest neighbor search, clustering).
- **Selection Algorithms in Machine Learning**: Applied in various scenarios for selecting important features or thresholding data.