Personal Algorithm Design Catalog
(CMSC 123 - Data Structures and Algorithms 2)

Table of Contents

I. Introduction

This personal algorithm catalog is a catalog of algorithms which covers six (6) algorithm design techniques. In each section, we will briefly discuss a particular design technique and an example algorithm.

Each algorithm is provided with a Java Program to help further understand how the algorithm works, together with notes to consider when user decides to run the program.

In this catalog, six (6) algorithm design techniques will be tackled, namely: Brute Force approach, Decrease-and-Conquer approach, Divide-and-Conquer approach, Dynamic Programming, Greedy approach, and Transform-and-Conquer approach. Under Brute Force, we will encounter its example algorithm: Selection Sort. Under Decrease-and-Conquer, we will go into another sorting algorithm: Insertion Sort. Under Divide-and-Conquer, we will discuss about three Binary Tree Traversals. Under Dynamic Programming, we will talk about an algorithm for all-pair shortest-path problem: Floyd-Warshall Algorithm. Under Greedy, we will deal with an algorithm that finds a minimum spanning tree: Prim's Algorithm. Finally, under Transform-and-Conquer, we will discuss an interesting sorting algorithm: Heapsort.

II. Brute Force

Brute force is a straightforward approach in solving a particular problem based on its statement and definition of concepts involved which considers every possible situation and relies on total computing power.

Brute force is applicable to a wide variety of problems, and it can serve an important educational purpose as a standard for comparison to find and judge more efficient approaches in solving a problem.

Example Algorithms include:

1. Selection Sort
2. Bubble Sort
3. Sequential Search

Selection Sort

Selection sort is an in-place comparison sorting algorithm wherein the list is divided into two parts – the sorted list on the left side, and the unsorted on the right side.

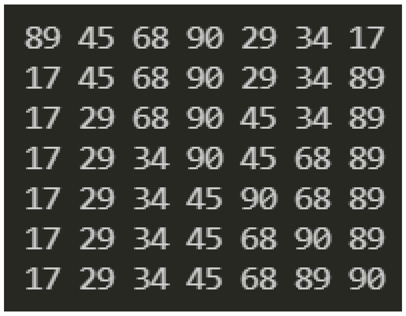
The time complexity of this algorithm is **$O(n^2)$** where n is the number of elements in the list.

See and run the program here:

AlgorithmDesignTechniques\BruteForce\SelectionSort\SelectionSort.java

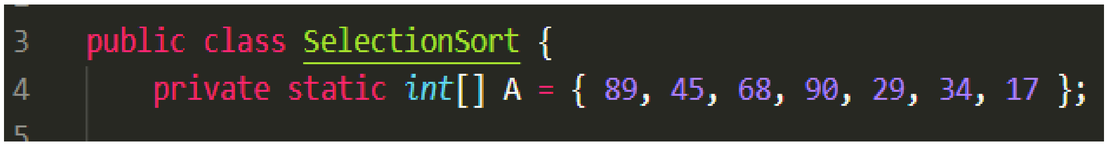
Notes:

1. Input is an unsorted array A of size n. Assume that each element of A is an integer.
2. Output is the sorted array A.
3. When user runs the program, the current sort of the array for each iteration will be displayed:



```
89 45 68 90 29 34 17
17 45 68 90 29 34 89
17 29 68 90 45 34 89
17 29 34 90 45 68 89
17 29 34 45 90 68 89
17 29 34 45 68 90 89
17 29 34 45 68 89 90
```

4. If user wishes to change input, modify array here:



```java
public class SelectionSort {
    private static int[] A = { 89, 45, 68, 90, 29, 34, 17 };
```

III. Decrease-and-Conquer

Decrease-and-Conquer technique is based on making use of the relationship (either top-down our bottom-up) between the solution of an instance of a problem and the solution to its smaller instance. It commonly leads to a recursive implementation when using the top-down variation and leads to an incremental approach when using the bottom-up variation.

Decrease-and-Conquer technique has 3 major variations:

(a) Decrease by a Constant – the size of an instance is reduced to a constant on each iteration,

(b) Decrease by a Constant Factor – a problem instance is reduced by the same constant factor, and

(c) Variable-size-decrease – the size-reduction pattern varies from one iteration to the other.

Decrease-and-Conquer technique is somewhat like Divide-and-Conquer approach, but instead of generating the problem to two or more subproblems, the problem is reduced to a single problem smaller than the original.

Example Algorithms include:

1. Insertion Sort
2. Topological Sorting
3. Binary Search
4. Interpolation Search

Insertion Sort

Insertion sort is a direct application of the decrease-(by one)-and-conquer technique to the sorting problem. The input array is divided into a sorted and an unsorted part, and the values from the latter are chosen and inserted at the correct position in the sorted part.

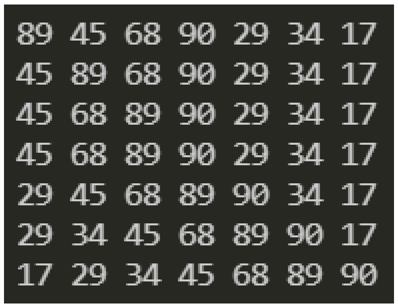
The time complexity of this algorithm is **O(n^2)** where n is the number of elements in the list.

See and run the program here:

AlgorithmDesignTechniques\DecreaseAndConquer\InsertionSort\InsertionSort.java

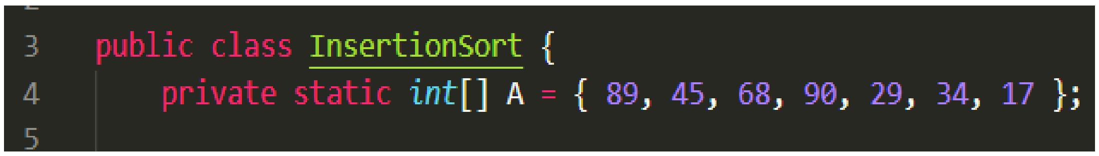
Notes:

1. Input is an unsorted array *A* of size *n*. Assume that each element of *A* is an integer.
2. Output is the sorted array *A*.
3. When user runs the program, the current sort of the array for each iteration will be displayed:



```
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
```

4. If user wishes to change input, modify array here:



```
public class InsertionSort {
    private static int[] A = { 89, 45, 68, 90, 29, 34, 17 };
```

IV. Divide-and-Conquer

The Divide-and-Conquer approach follows a general plan: A problem is divided into subproblems of the same type, each are solved recursively, then the solutions to the subproblems are combined to get a solution to the original problem.

Divide-and-Conquer approach is suitable for problems that can be solved by parallel computing – subproblems are solved simultaneously by its own processor.

Example Algorithms include:

1. Merge sort
2. Quick Sort
3. Binary Tree Traversals

    Preorder;

    Inorder; and

    Postorder
4. Strassen's Algorithm

Binary Tree Traversals

Binary Tree Traversals are the most important divide-and-conquer algorithms for Binary Trees. These traversals visit the nodes of the binary tree recursively by visiting the root and the left and right subtrees.

There are three binary tree traversals based on the order of visiting:

Preorder traversal – the root is visited first, then the left and right subtrees, respectively. (Root – Left – Right)

Inorder traversal – the left subtree is visited first, then the root and the right subtree, respectively. (Left – Root – Right)

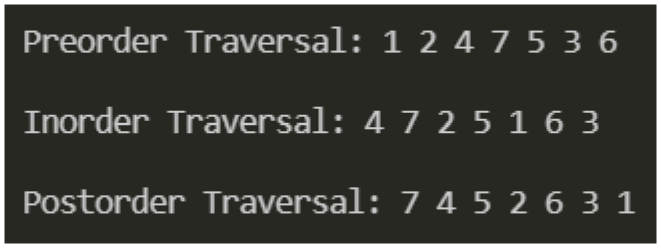
Postorder traversal – the left subtree is visited first, then the right subtree and the root, respectively. (Left – Right – Root)

The time complexity of these algorithms is **O(n)** where n is the number of nodes in the binary tree.

See and run the program here: AlgorithmDesignTechniques\DivideAndConquer\BinaryTreeTraversals\BinaryTreeTraversals.java

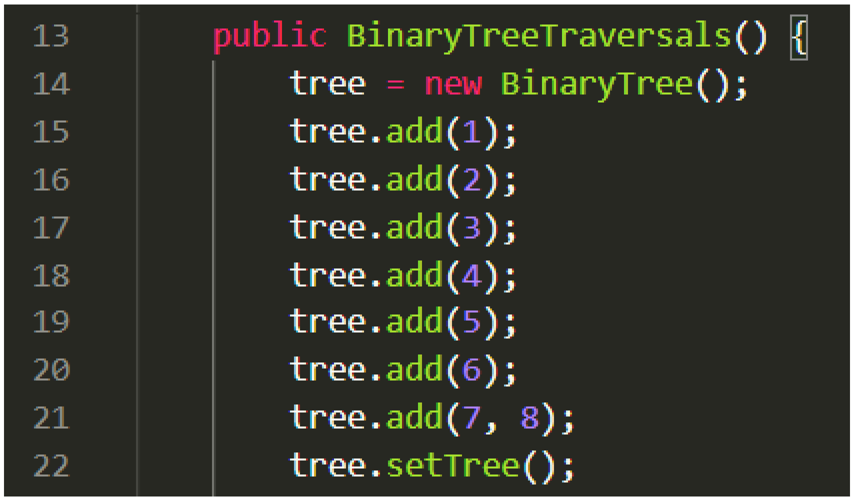
Notes:

1. Input is a binary tree. Assume that each node is labeled according to its value (integer).
2. Output is a sequence of nodes visited during the traversal:



```
Preorder Traversal: 1 2 4 7 5 3 6

Inorder Traversal: 4 7 2 5 1 6 3

Postorder Traversal: 7 4 5 2 6 3 1
```

3. If user wishes to change input, modify tree here:



```
13    public BinaryTreeTraversals() {
14        tree = new BinaryTree();
15        tree.add(1);
16        tree.add(2);
17        tree.add(3);
18        tree.add(4);
19        tree.add(5);
20        tree.add(6);
21        tree.add(7, 8);
22        tree.setTree();
```

The function *add* with two parameters (e.g. "tree.add(7 , 8)" in line 21) asks for the value of the node and its position in the binary tree.

V. Dynamic Programming

Dynamic Programming is a technique for solving problems with overlapping subproblems. Instead of repeatedly solving these subproblems, the technique suggests performing the solution to a smaller problem once, then recording the result in a table from which a solution to the original problem can be obtained.

Dynamic Programming is best used when problems satisfy the **principle of optimality** (the problem is divided to subproblems, and their results can be reused).

Example Algorithms include:

1. Solving the change-making problem by dynamic programming
2. Solving the knapsack problem by dynamic programming
3. Construction of Optimal Binary Search Tree
4. Warshall's Algorithm
5. Floyd's Algorithm

Floyd-Warshall Algorithm

The Floyd-Warshall Algorithm solves the all-pairs shortest-path problem in a weighted graph. The input for this algorithm is a weighted graph with n vertices (assume that no vertex has a loop, and two vertices can only have at most one edge), and the output is an n x n matrix (implemented as a 2D

Array) which shows the shortest path from one vertex to every other vertex in a graph.

ALGORITHM:

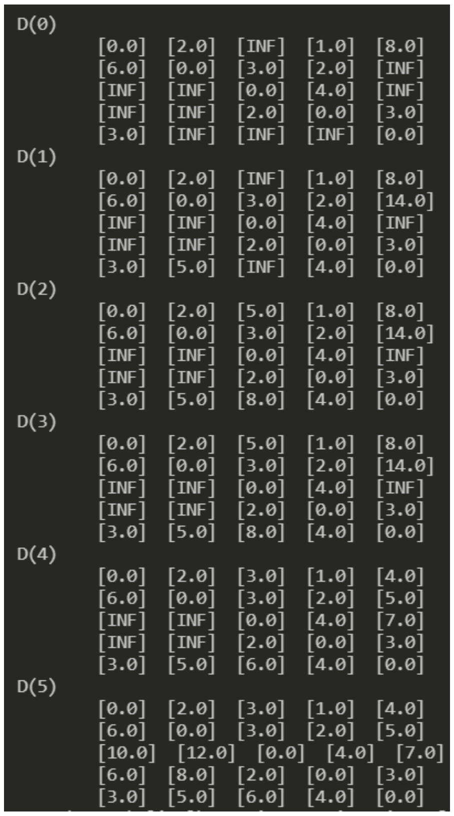
```
let dist[][] = n x n array of minimum distances


for each vertex v, do
        dist[v][v] = 0
for each edge (f, s), do
        dist[f][s] = weight of edge (f, s)


for k from 1 to n
        for i from 1 to n
                for j from 1 to n
                        if dist[i][j] > dist[i][k] + dist[k][j], then
                                dist[i][j] = dist[i][k] + dist[k][j]
```

The time complexity of this algorithm is **$O(n^3)$** where n is the number of vertices in the graph.

See and run the program here:
AlgorithmDesignTechniques\DynamicProgramming\FloydWarshallAlgorithm\FloydWarshallAlgorithm.java

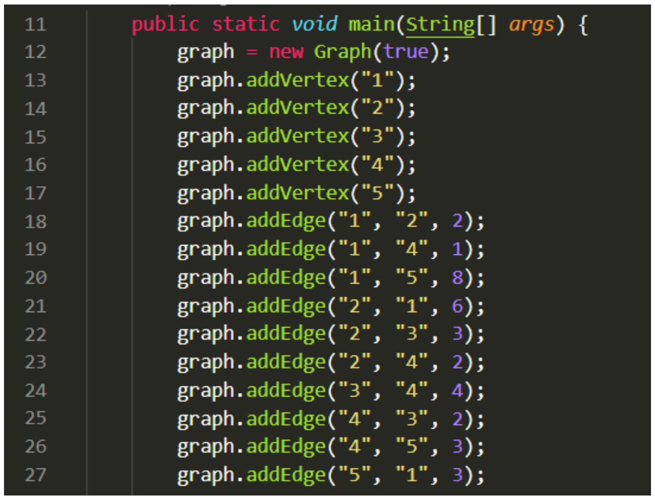
Notes:

1. When user runs the program, an update on the n x n matrix on every iteration will be displayed:



```
D(0)
        [0.0]  [2.0]  [INF]  [1.0]  [8.0]
        [6.0]  [0.0]  [3.0]  [2.0]  [INF]
        [INF]  [INF]  [0.0]  [4.0]  [INF]
        [INF]  [INF]  [2.0]  [0.0]  [3.0]
        [3.0]  [INF]  [INF]  [INF]  [0.0]
D(1)
        [0.0]  [2.0]  [INF]  [1.0]  [8.0]
        [6.0]  [0.0]  [3.0]  [2.0]  [14.0]
        [INF]  [INF]  [0.0]  [4.0]  [INF]
        [INF]  [INF]  [2.0]  [0.0]  [3.0]
        [3.0]  [5.0]  [INF]  [4.0]  [0.0]
D(2)
        [0.0]  [2.0]  [5.0]  [1.0]  [8.0]
        [6.0]  [0.0]  [3.0]  [2.0]  [14.0]
        [INF]  [INF]  [0.0]  [4.0]  [INF]
        [INF]  [INF]  [2.0]  [0.0]  [3.0]
        [3.0]  [5.0]  [8.0]  [4.0]  [0.0]
D(3)
        [0.0]  [2.0]  [5.0]  [1.0]  [8.0]
        [6.0]  [0.0]  [3.0]  [2.0]  [14.0]
        [INF]  [INF]  [0.0]  [4.0]  [INF]
        [INF]  [INF]  [2.0]  [0.0]  [3.0]
        [3.0]  [5.0]  [8.0]  [4.0]  [0.0]
D(4)
        [0.0]  [2.0]  [3.0]  [1.0]  [4.0]
        [6.0]  [0.0]  [3.0]  [2.0]  [5.0]
        [INF]  [INF]  [0.0]  [4.0]  [7.0]
        [INF]  [INF]  [2.0]  [0.0]  [3.0]
        [3.0]  [5.0]  [6.0]  [4.0]  [0.0]
D(5)
        [0.0]  [2.0]  [3.0]  [1.0]  [4.0]
        [6.0]  [0.0]  [3.0]  [2.0]  [5.0]
        [10.0]  [12.0]  [0.0]  [4.0]  [7.0]
        [6.0]  [8.0]  [2.0]  [0.0]  [3.0]
        [3.0]  [5.0]  [6.0]  [4.0]  [0.0]
```

2. If user wishes to change the input, modify the tree here:



```java
    public static void main(String[] args) {
        graph = new Graph(true);
        graph.addVertex("1");
        graph.addVertex("2");
        graph.addVertex("3");
        graph.addVertex("4");
        graph.addVertex("5");
        graph.addEdge("1", "2", 2);
        graph.addEdge("1", "4", 1);
        graph.addEdge("1", "5", 8);
        graph.addEdge("2", "1", 6);
        graph.addEdge("2", "3", 3);
        graph.addEdge("2", "4", 2);
        graph.addEdge("3", "4", 4);
        graph.addEdge("4", "3", 2);
        graph.addEdge("4", "5", 3);
        graph.addEdge("5", "1", 3);
```

The function *addEdge* with three parameters (e.g. "graph.addEdge("1", "2", 2)") asks for the first vertex (String), the second vertex (String), and the weight (double), respectively.

VI. Greedy

The goal of the Greedy Technique is to construct a solution through a sequence of steps, which creates a partially constructed solution, until a completed solution is reached. On each step, a choice needs to satisfy the following criteria:

1. feasible (the constraints of the problem are satisfied),
2. locally optimal (the choice must be the best among the available choices), and
3. irrevocable (once the choice has been made, it cannot be changed during the succeeding steps).

Using Greedy Technique is best when a problem satisfies two properties:

(a) The Greedy-choice Property, in which the global optimum can be reached by selecting a local optimum, and

(b) The Optimal Substructure, whose optimal solution contains optimal solutions to its subproblems.

Example Algorithms include:

1. Prim's Algorithm
2. Kruskal's Algorithm
3. Dijkstra's algorithm
4. Huffman codes

Prim's Algorithm

Prim's Algorithm is a greedy algorithm for constructing a minimum spanning tree of a weighted connected graph which works by attaching to a previously constructed subtree a vertex closest to the vertices already in the tree. The input for this algorithm is a weighted graph with n vertices (assume that no vertex has a loop, and two vertices can only have at most one edge), and the output is a set of edges which forms a minimum spanning tree.

The time complexity of this algorithm depends on the data structure used in the algorithm:

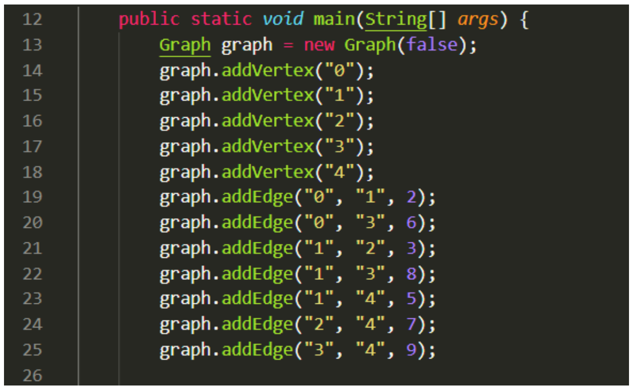
> If the data structure used is the Adjacency Matrix Searching, then the time complexity is **$O(V^2)$** where V is the number of vertices in the graph.
>
> If the data structure used is the Binary Heap or Adjacency List, then the time complexity is **$O(V \log V + E \log V)$** where V is the number of vertices and E is the number of edges.

See and run program here:

AlgorithmDesignTechniques\Greedy\PrimAlgorithm\PrimAlgorithm.java

Notes:

1. If user wishes to change the input, modify the graph here:



```
    public static void main(String[] args) {
        Graph graph = new Graph(false);
        graph.addVertex("0");
        graph.addVertex("1");
        graph.addVertex("2");
        graph.addVertex("3");
        graph.addVertex("4");
        graph.addEdge("0", "1", 2);
        graph.addEdge("0", "3", 6);
        graph.addEdge("1", "2", 3);
        graph.addEdge("1", "3", 8);
        graph.addEdge("1", "4", 5);
        graph.addEdge("2", "4", 7);
        graph.addEdge("3", "4", 9);

```

The function *addEdge* with three parameters (e.g. "graph.addEdge("0", "1", 2)") asks for the first vertex (String), the second vertex (String), and the weight (double), respectively.

VII. Transform-and-Conquer

Transform-and-Conquer is an approach which deals with two-stage procedures:

1. The Transformation Stage, where the instance of a problem is modified to be more amenable to solution, and
2. The Conquering Stage, where it is then solved.

Transform-and-Conquer approach is best if a problem's instance can be transformed to one of three major variants:

1. Instance Simplification (Transform to a simpler instance of the same problem)
2. Representation Change (Transform to another representation)
3. Problem Reduction (Transform to an instance of a different problem with an available algorithm)

Example Algorithms include:

For Instance Simplification:

1. List Presorting;
2. Gaussian Elimination (an algorithm for solving systems of linear equations by transforming it to an equivalent system with an upper-triangular coefficient matrix); and
3. Rotations in AVL Tree

For Representation Change:

4. Representation of a set by a 2-3 tree;

5. Heapsort (sorting algorithm based on arranging elements of an array in a heap and then successively removing the largest element from a remaining heap); and

6. Horner's rule for polynomial evaluation (an optimal algorithm for polynomial evaluation without coefficient preprocessing, which requires only n multiplications and n additions to evaluate an n-degree polynomial at a given point)

For Problem Reduction

7. Computing the least common multiple;

8. Counting paths in a graph;

9. Reduction of Optimization Problems;

10. Linear Programming; and

11. Reduction to Graph Problems

Heapsort

Heapsort is a sorting algorithm with two stages:

1. Construction of a Binary Heap
2. Delete root by replacing it with last element, reduce the heap by one, heapify root, then repeat step n-1 times (where n is number of nodes in binary heap)

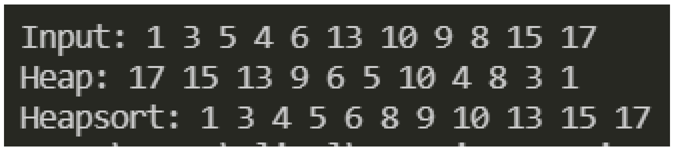
Construction of the binary heap has a time complexity of **O(n)** and the heapify function has a time complexity of **O(log n)**. Overall, the time complexity of this algorithm is **O(n log n)**.

See and run program here:

AlgorithmDesignTechniques\TransformAndConquer\HeapSort\HeapSort.java

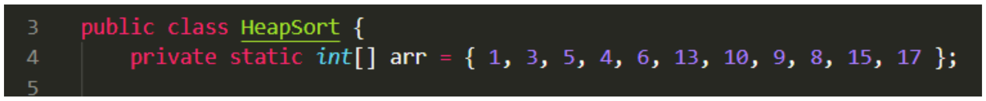
Notes:

1. The Binary Heap is implemented as an array, thus input is an array.
2. When user runs the program, the sequence of elements prior to construction of heap, after construction of heap, and after heapsort will be displayed:



```
Input: 1 3 5 4 6 13 10 9 8 15 17
Heap: 17 15 13 9 6 5 10 4 8 3 1
Heapsort: 1 3 4 5 6 8 9 10 13 15 17
```

3. If user wishes to change the elements, modify the array here:



```
public class HeapSort {
    private static int[] arr = { 1, 3, 5, 4, 6, 13, 10, 9, 8, 15, 17 };

```