Applications of Decrease and Conquer

1. Introduction

Decrease & conquer is a general algorithm design strategy based on exploiting the relationship between
a solution to a given instance of a problem and a solution to a smaller instance of the same problem.
The exploitation can be either top-down (recursive) or bottom-up (non-recursive).
The major variations of decrease and conquer are

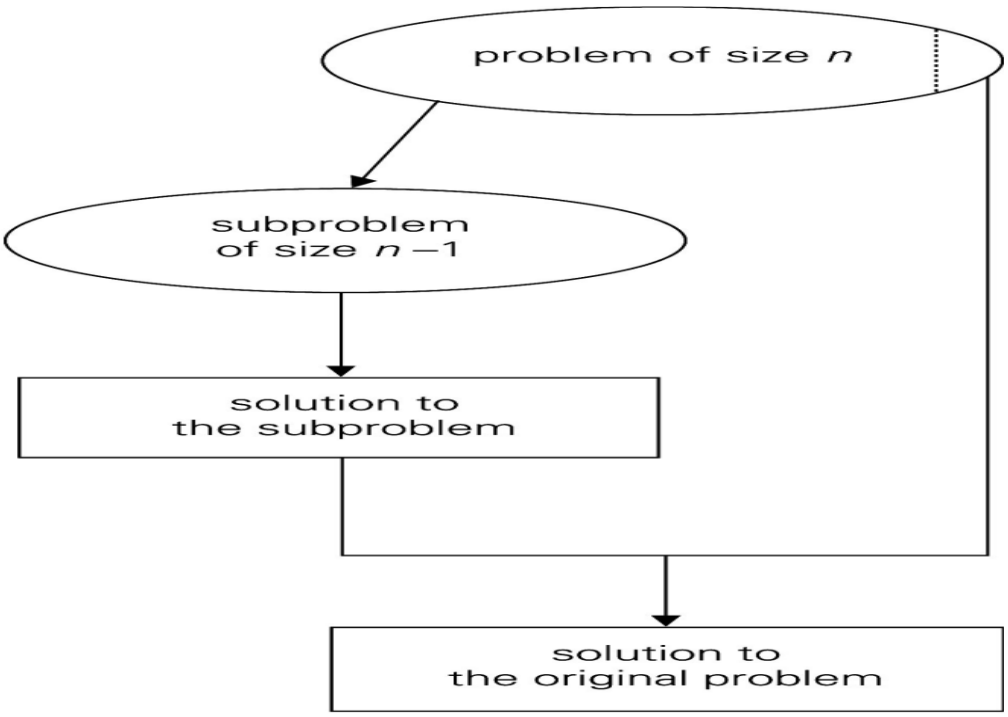
1. Decrease by a constant :(usually by 1):
a. insertion sort
b. graph traversal algorithms (DFS and BFS)
c. topological sorting
d. algorithms for generating permutations, subsets.

2. Decrease by a constant factor (usually by half)
a. binary search and bisection method

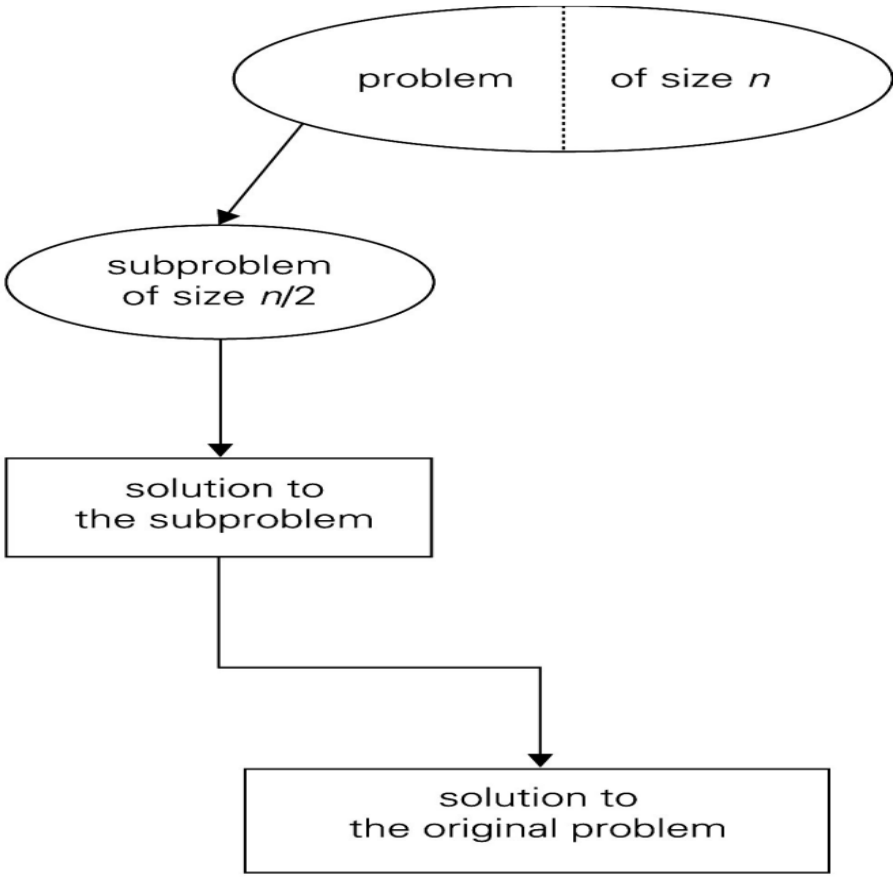
3. Variable size decrease
a. Euclid‘s algorithm

Following diagram shows the major variations of decrease & conquer approach.

Decrease by a constant :(usually by 1):



Decrease by a constant factor (usually by half)



2. Insertion sort

Insertion sort is an application of decrease & conquer technique. It is a comparison based sort in which the sorted array is built on one entry at a time.

$$A[0] \le \ldots \le A[j] < A[j+1] \le \ldots \le A[i-1] \mid A[i] \ldots A[n-1]$$

smaller than or equal to $A[i]$ greater than $A[i]$

Algorithm:

ALGORITHM Insertionsort(A [0 … n-1])
//sorts a given array by insertion sort
//i/p: Array A[0…n-1]
//o/p: sorted array A[0…n-1] in ascending order

```
for i    →    1 to n-1
                V→     A[i]
                j→     i-1
                while j ≥ 0 AND A[j] > V do
                        A[j+1]     A[j]
                        j→     j – 1
                A[j + 1]→      V
```

Analysis:
• **Input size:** Array size, n
• **Basic operation:** key comparison
• Best, worst, average case exists
Best case: when input is a sorted array in ascending order:
Worst case: when input is a sorted array in descending order:
Let $C_{worst}(n)$ be the number of key comparison in the worst case. Then

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Let $C_{best}(n)$ be the number of key comparison in the best case.
Then

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

Example:
Sort the following list of elements using insertion sort:

89, 45, 68, 90, 29, 34, 17
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

3. Depth-first search (DFS) and Breadth-first search (BFS)

DFS and BFS are two graph traversing algorithms and follow decrease and conquer approach – decrease by one variation to traverse the graph.

Some useful definition:

• **Tree edges:** edges used by DFS traversal to reach previously unvisited vertices
• **Back edges:** edges connecting vertices to previously visited vertices other than their immediate predecessor in the traversals.
• **Cross edges:** edge that connects an unvisited vertex to vertex other than its immediate predecessor. (connects siblings)
• **DAG:** Directed acyclic graph.

Depth-first search (DFS)
Description:

• DFS starts visiting vertices of a graph at an arbitrary vertex by marking it as visited.
• It visits graph‘s vertices by always moving away from last visited vertex to an unvisited one, backtracks if no adjacent unvisited vertex is available.
• Is a recursive algorithm, it uses a stack
• A vertex is pushed onto the stack when it‘s reached for the first time
• A vertex is popped off the stack when it becomes a dead end, i.e., when there is no adjacent unvisited vertex
• ―Redraws‖ graph in tree-like fashion (with tree edges and back edges for undirected graph).

Algorithm:

ALGORITHM DFS (G)

//implements DFS traversal of a given graph
//i/p: Graph G = { V, E}
//o/p: DFS tree
Mark each vertex in V with 0 as a mark of being "unvisited"
count→ 0
for each vertex v in V do
    if v is marked with 0
        dfs(v)

dfs(v)

count→ count + 1
mark v with count
for each vertex w in V adjacent to v do
    if w is marked with 0
        dfs(w)

Example:

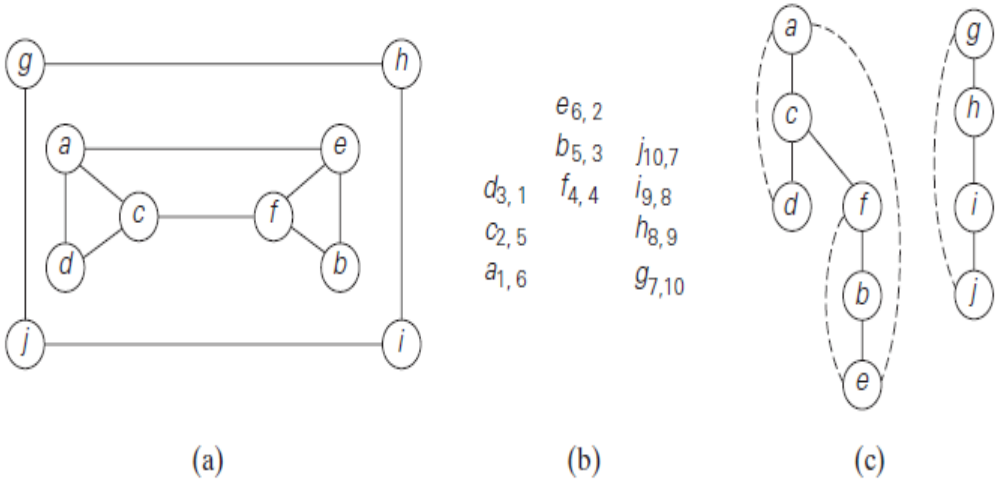


FIGURE 3.10 Example of a DFS traversal. (a) Graph. (b) Traversal's stack (the first subscript number indicates the order in which a vertex is visited, i.e., pushed onto the stack; the second one indicates the order in which it becomes a dead-end, i.e., popped off the stack). (c) DFS forest with the tree and back edges shown with solid and dashed lines, respectively.

Applications of DFS:

• The two orderings are advantageous for various applications like topological sorting, etc

• To check connectivity of a graph (number of times stack becomes empty tells the number of components in the graph)

• To check if a graph is acyclic. (no back edges indicates no cycle)

• To find articulation point in a graph

Efficiency:

• Depends on the graph representation:

o Adjacency matrix : Θ(n2)

o Adjacency list: Θ(n + e)

Breadth-first search (BFS)

Description:

• BFS starts visiting vertices of a graph at an arbitrary vertex by marking it as visited.

• It visits graph‘s vertices by across to all the neighbours of the last visited vertex.

• Instead of a stack, BFS uses a queue.

• Similar to level-by-level tree traversal.

• ―Redraws‖ graph in tree-like fashion (with tree edges and cross edges for undirected graph).

Algorithm:

ALGORITHM BFS (G)

```
//implements BFS traversal of a given graph
//i/p: Graph G = { V, E}
//o/p: BFS tree/forest
Mark each vertex in V with 0 as a mark of being "unvisited"
count→      0
for each vertex v in V do
        if v is marked with 0
                bfs(v)
```

bfs(v)

```
count→     count + 1
mark v with count and initialize a queue with v
while the queue is NOT empty do
        for each vertex w in V adjacent to front's vertex v do
                if w is marked with 0
                        count→     count + 1
                        mark w with count
                        add w to the queue
                remove vertex v from the front of the queue
```

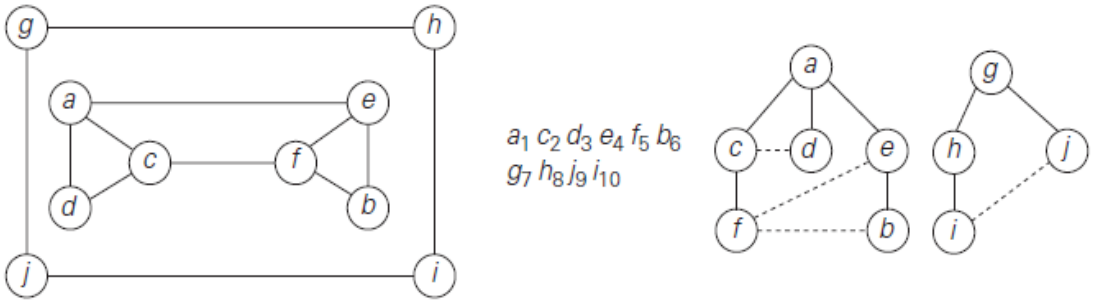



FIGURE 3.11 Example of a BFS traversal. (a) Graph. (b) Traversal queue, with the numbers indicating the order in which the vertices are visited, i.e., added to (and removed from) the queue. (c) BFS forest with the tree and cross edges shown with solid and dotted lines, respectively.

Applications of BFS:

- To check connectivity of a graph (number of times queue becomes empty tells the number of components in the graph)
- To check if a graph is acyclic. (no cross edges indicates no cycle)
- To find minimum edge path in a graph

Efficiency:

- Depends on the graph representation:
  - Array : $\Theta(n^2)$
  - List: $\Theta(n+e)$

TABLE 3.1 Main facts about depth-first search (DFS) and breadth-first search (BFS)

	DFS	BFS
Data structure	a stack	a queue
Number of vertex orderings	two orderings	one ordering
Edge types (undirected graphs)	tree and back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, minimum-edge paths
Efficiency for adjacency matrix	$\Theta(\|V^2\|)$	$\Theta(\|V^2\|)$
Efficiency for adjacency lists	$\Theta(\|V\|+\|E\|)$	$\Theta(\|V\|+\|E\|)$