Decrease and Conquer

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

Also referred to as inductive or incremental approach
I have also seen this called “Chip and Conquer”

Examples of Decrease and Conquer

Decrease by one:
- Insertion sort
- Graph search algorithms:
  - DFS
  - BFS
  - Topological sorting
- Algorithms for generating permutations, subsets

Decrease by a constant factor
- Binary search
- Fake-coin problems
- Multiplication à la russe
- Josephus problem

Variable-size decrease
- Euclid’s algorithm
- Selection by partition

Insertion Sort

Strategy:
- Insertion of an element in proper order:
  - Begin with a sequence E of n elements in arbitrary order
  - Initially assume the sorted segment contains first element
  - Let x be the next element to be inserted in sorted segment, pull x “out of the way”, leaving a vacancy
  - repeatedly compare x to the element just to the left of the vacancy, and as long as x is smaller, move that element into the vacancy,
  - else put x in the vacancy,
  - repeat the next element that has not yet examined.

Example created by Dr. Cusack (2002)
Insertion Sort: Specification for subroutine

- Specification
  - int shiftVacRec(Element[] E, int vacant, Key x)
  - Precondition
    - vacant is nonnegative
  - Postconditions
    - 1. Elements in E at indices less than xLoc are in their original positions and have keys less than or equal to x.
    - 2. Elements in E at positions xLoc+1, ..., vacant are greater than x and were shifted up by one position from their positions when shiftVacRec was invoked.

Insertion Sort: Algorithm shiftVacRec

```c
int shiftVacRec(Element[] E, int vacant, Key x)
```

```c
int xLoc;
if (vacant == 0)
xLoc = vacant;
else if (E[vacant-1].key <= x)
xLoc = vacant;
else
E[vacant] = E[vacant-1];
xLoc = shiftVacRec(E, vacant-1, x);
return xLoc
```

Insertion Sort: Analysis

- Worst-Case Complexity
- Average Behavior
  - average number of comparisons in shiftVacRec
  - Thus,

Insertion Sort: Loop Implementation

```c
Sample C++ code for insertion sort:

```c
void insertionSort(int A[], int n) {
    for (int j = 1; j < n; j++) {
        int temp = A[j];
        while (j > 0 && temp < A[j-1]) {
            A[j] = A[j-1];
            j--;
        }
        A[j] = temp;
    }
```

Graph Traversal

- Many problems require processing all graph vertices in systematic fashion
- **Graph traversal algorithms:**
  - Depth-first search
  - Breadth-first search
- **First, some definitions!**
A (simple) graph $G = (V, E)$ consists of
- $V$, a nonempty set of vertices
- $E$, a set of unordered pairs of distinct vertices called edges.

Examples:

$V = \{A, B, C, D, E\}$
$E = \{(A, D), (A, E), (B, D), (B, E), (C, D), (C, E)\}$

A weighted graph is a triple $G = (V, E, W)$
- where $(V, E)$ is a graph (or a digraph) and
- $W$ is a function from $E$ into $R$, the reals (integer or rationals).
- For an edge $e$, $W(e)$ is called the weight of $e$.

A weighted digraph is often called a network.

Examples

Let $u$ and $v$ be vertices, and let $e = (u, v)$ be an edge in an undirected graph $G$.
- The vertices $u$ and $v$ are adjacent.
- The edge $e$ is incident with both vertices $u$ and $v$.
- The edge $e$ connects $u$ and $v$.
- The vertices $u$ and $v$ are the endpoints of edge $e$.
- The degree of a vertex, denoted $\deg(v)$, in an undirected graph is the number of edges incident with it (where self-loops are counted twice).

More Graph Terminology

A subgraph of a graph $G = (V, E)$ is a graph $G = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$.

A path is a sequence of vertices $v_1, v_2, v_3, \ldots, v_k$ such that consecutive vertices $v_i$ and $v_{i+1}$ are adjacent.

(a simple path)
Design and Analysis of Algorithms

Chapter 5

Graph Representations using Data Structures

- Adjacency Matrix Representation
  - Let $G = (V, E, \alpha = |V|, \beta = |E|, \mathsf{E} = \{e_1, e_2, \ldots, e_\beta\})$
  - $G$ can be represented by an $\alpha \times \alpha$ matrix.

(1) An undirected graph
(2) Its adjacency matrix

(3) A weighted digraph
(4) Its adjacency matrix

Adjacency List Representation

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

(1) A non-null directed graph
(2) Its adjacency list

Weighted Digraph Adjacency List Representation

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

(1) A directed graph
(2) Its adjacency list

More Definitions

- Subgraph
- Symmetric digraph
- Complete graph
- Adjacency relation
- Path, simple path, reachable
- Connected, Strongly Connected
- Cycle, simple cycle
- Acyclic
- Undirected forest
- Free tree, undirected tree
- Rooted tree
- Connected component

Traversing Graphs

- Most algorithms for solving problems on a graph examine or process each vertex and each edge.
- Depth-First Search (DFS) and Breadth-First Search (BFS)
  - Two elementary traversal strategies that provide an efficient way to “visit” each vertex and edge exactly once.
  - Both work on directed or undirected graphs.
  - Many advanced graph algorithms are based on the concepts of DFS or BFS.
  - The difference between the two algorithms is in the order in which each “visits” vertices.
**Depth-first search**

- Explore graph always moving away from last visited vertex.
- Similar to preorder tree traversals.

Pseudocode for Depth-first-search of graph \( G = (V, E) \):

```plaintext
DFS(G)
    count := 0
    for each vertex \( v \) do
        mark \( v \) with count
    for each vertex \( v \) do
        if \( v \) is marked with 0
            dfs(v)
```

**Example – undirected graph**

```
Example – undirected graph
```

**Types of edges**

- **Tree edges**: edges comprising forest.
- **Back edges**: edges to ancestor nodes.
- **Forward edges**: edges to descendants (digraphs only).
- **Cross edges**: none of the above.

**Example – directed graph**

```
Example – directed graph
```

**Depth-first search: Notes**

- DFS can be implemented with graphs represented as:
  - Adjacency matrices: \( O(V^2) \)
  - Adjacency linked lists: \( O(V + E) \)
- Yields two distinct ordering of vertices:
  - preorder: as vertices are first encountered (pushed onto stack)
  - postorder: as vertices become dead-ends (popped off stack)
- Applications:
  - Checking connectivity, finding connected components
  - Checking acyclicity
  - Searching state-space of problems for solution (AI)

**Breadth-first search**

- Explore graph moving across to all the neighbors of last visited vertex.
- Similar to level-by-level tree traversals.
- Instead of a stack, breadth-first uses queue.
- Applications: same as DFS, but can also find paths from a vertex to all other vertices with the smallest number of edges.
Breadth-first search algorithm

BFS(G)
count := 0
mark each vertex with 0
for each vertex \( v \in V \) do
  bfs(v)

bfs(v)
count := count + 1
mark \( v \) with count
initialize queue with \( v \)
while queue is not empty do
  a := front of queue
  for each vertex \( w \) adjacent to \( a \) do
    if \( w \) is marked with 0
      count := count + 1
      mark \( w \) with count
      add \( w \) to the end of the queue
  remove \( a \) from the front of the queue

Example – undirected graph

Example – directed graph

Breadth-first search: Notes

- BFS has same efficiency as DFS and can be implemented with graphs represented as:
  - Adjacency matrices: \( \Theta(V^2) \)
  - Adjacency linked lists: \( \Theta(V + E) \)

- Yields single ordering of vertices (order added/deleted from queue is the same)

Directed acyclic graph (dag)

- A directed graph with no cycles
- Arise in modeling many problems, eg:
  - prerequisite structure
  - food chains
- Imply partial ordering on the domain

Topological sorting

- Problem: find a total order consistent with a partial order
- Example:
  - Order them so that they don’t have to wait for any of their food
  - i.e., from lower to higher, consistent with food chain
  - NB: problem is solvable iff graph is dag
### Topological sorting Algorithms

1. **DFS-based algorithm:**
   - DFS traversal noting order vertices are popped off stack
   - Reverse order solves topological sorting
   - Back edges encountered? → NOT a dag!

2. **Source removal algorithm**
   - Repeatedly identify and remove a source vertex, i.e., a vertex that has no incoming edges

- Both \( \Theta(V+E) \) using adjacency linked lists

### Binary Search Trees (BSTs)

- **Binary Search Tree property**
  - A binary tree in which the key of an internal node is greater than the keys in its left subtree and less than or equal to the keys in its right subtree.

- An inorder traversal of a binary search tree produces a sorted list of keys.

### Variable-size-decrease: Binary search trees

- Keys are arranged in a binary tree with the **binary search tree property:**

  ![](image1)

  - What about repeated keys?

### BST Examples

- Binary Search trees with different degrees of balance
- Black dots denote empty trees

### BST Operations

- Find the min/max element
- Search for an element
- Find the successor/predecessor of an element
- Insert an element
- Delete an element
Finding the successor of a node

- For a tree as balanced as possible, \( \Theta(\lg n) \)
  - Why?
- The objective is to make the tree as balanced as possible
  - Technique: Binary Tree Rotations (more on this later)

BST: Successor/Predecessor

- Finding the successor of a node \( x \) (if it exists):
  - If rightSubtree(\( x \)) is empty, then successor(\( x \)) is the lowest ancestor of \( x \) whose left child is also an ancestor of \( x \).

Analysis of Binary Search Tree Retrieval

- Let \( n \) be the number of internal nodes of the tree that are examined while searching for key
- For a long chain tree structure, \( \Theta(n) \)
- For a tree as balanced as possible, \( \Theta(\lg n) \)
  - Why?
- The objective is to make the tree as balanced as possible
  - Technique: Binary Tree Rotations (more on this later)

BST Search (Retrieval)

```cpp
Element bstSearch(BinTree bst, Key K) {
  1. if (bst == nil) {
      2. found = bstSearch(leftSubtree(bst), K);
      3. if (rightSubtree(bst) != nil) {
          4. found = bstSearch(rightSubtree(bst), K);
      }
      5. return found;
  }
  6. if (K < bst.key) {
      7. return bstSearch(leftSubtree(bst), K);
  }
  8. else if (K > bst.key) {
      9. return bstSearch(rightSubtree(bst), K);
  }
  10. return bstSearch(leftSubtree(bst), K);
  11. return bstSearch(rightSubtree(bst), K);
}
```

BST: Successor/Predecessor

- The objective is to make the tree as balanced as possible
- Technique: Binary Tree Rotations (more on this later)

Element bstSearch(BinTree bst, Key K)

1. Element found
2. if (bst == nil)
3. found = null;
4. else
5. Element root = root(bst);
6. if (K == root.key)
7. found = root;
8. else if (K < root.key)
9. found = bstSearch(leftSubtree(bst), K);
10. else
11. found = bstSearch(rightSubtree(bst), K);
12. return found;

BST: Successor/Predecessor

- Finding the successor of a node \( x \) (if it exists):
  - If rightSubtree(\( x \)) is empty, then successor(\( x \)) is the smallest element in the tree root at rightSubtree(\( x \))
    - Why?

- The predecessor operation is symmetric to successor.
- Try defining it.
- Write code for each operation.
Design and Analysis of Algorithms - Chapter 5

**BST: Insertion**

- To insert a node into a BST, we search the tree until we find a node whose appropriate subtree is empty, and insert the new node there.

**BST: Delete**

- Deleting a node \( z \) is by far the most difficult BST operation.
- There are three cases to consider:
  - If \( z \) has no children, just delete it.
  - If \( z \) has one child, splice out \( z \). That is, link \( z \)'s parent and child.
  - If \( z \) has two children, splice out \( z \)'s successor \( y \), and replace the contents of \( z \) with the contents of \( y \).
- The last case works because if \( z \) has two children, then its successor has no left child.

**BST:splice out algorithm**

- tail works when a node has at most the child

**BST: Deletion Algorithm**

- Delete is now simple!

  ```java
  Delete(T,x) {
    if x.left NIL || x.right NIL) 
     SpliceOut(T,x); 
    else { 
      y = successor(x); 
      x->key = y->key; 
      y->key = y->key 
      SpliceOut(T,y); 
    }
  }
  ```
BST: delete examples

BST: one more delete example

Analysis of BST Operations

- All of the BST operations have time complexity $O(h)$, where $h$ is the height of the tree
- However, in the worst-case, the height may be $O(n)$ where $n$ is the number of internal nodes
  - For example, a long chain tree structure
- For a tree as balanced as possible, $\Omega(\log n)$
  - Why?
- The objective is to make the tree as balanced as possible
  - Technique: Binary Tree Rotations
- 2-3 Trees and Red-Black Trees are BSTs that have height $\Theta(\log n)$