

# CS 395 – Analysis of Algorithms

## Chapter 4 – Decrease-and-Conquer

### Read 4.1

- **Decrease-and-Conquer techniques**
  - General idea
  - Three types of Decrease-n-Conquer algorithms
- **Insertion Sort**
- **Shellsort (reading assignment)**

# Decrease-and-Conquer



Ω Based on exploiting relationship between a solution to the given instance of a problem and that to a **smaller instance**

- **Reduce** problem instance to smaller instance of the same problem
- **Solve** smaller instance
- **Extend** solution of smaller instance to obtain solution to original instance

Ω Can be implemented either

- **bottom-up** (usually implemented iteratively) or
- **top-down** (usually implemented using recursion)

Ω Also referred to as *inductive* or *incremental* approach



## 3 Types of Decrease and Conquer

### ⌘ Decrease by a constant (usually by 1):

- insertion sort
- topological sorting
- algorithms for generating permutations, subsets

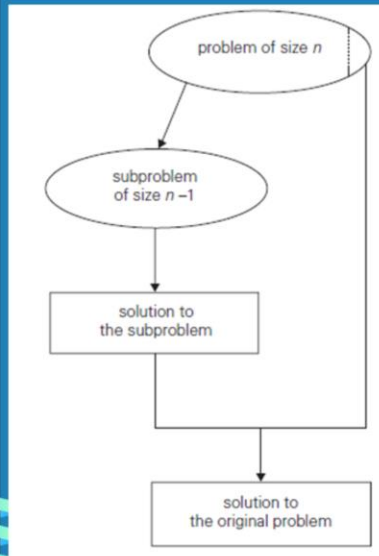
### ⌘ Decrease by a constant factor (usually by half)

- binary search and bisection method
- exponentiation by squaring
- multiplication à la russe

### ⌘ Variable-size decrease

- Euclid's algorithm
- selection by partition
- Nim-like games

# Decrease-(by one)-and-Conquer



For example, computing  $a^n$   
 $a^n = a^{n-1} \cdot a$

**Top-down:**

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

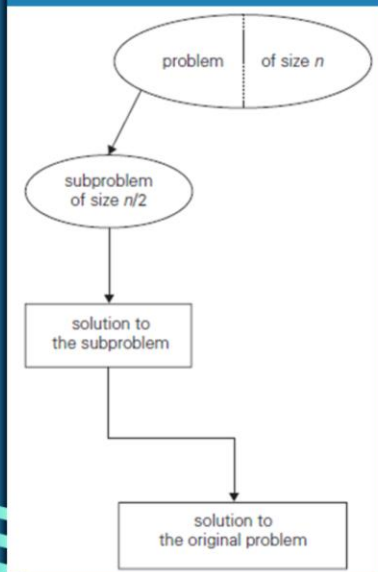
**Bottom-up:**

multiply **1** by  $a$   $n$  times.

- Similar to brute-force
- ... but coming from a different thought process



# Decrease-(by half)-and-Conquer



For example, computing  $a^n$  (again)

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

Top-down:

- Recursive implementation
- Efficiency:  $\Theta(\log n)$

5

If the instance of size  $n$  is to compute  $a^n$ , the instance of half its size is to compute  $a^{n/2}$ , with the obvious

$$a^n = (a^{n/2})^2.$$

But this does not work for odd  $n$ .

If  $n$  is odd, we have to compute  $a^{n-1}$  by using the rule for even-valued exponents and then multiply the

## Decrease-(by variable-size)-and-Conquer

Ω Size-reduction pattern varies from one iteration to another

Ω Example: Euclid's algorithm for GCD

- $\text{gcd}(m, n) = \text{gcd}(n, m \bmod n)$
- Second argument decreasing at variable rate



# Insertion Sort

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  $A[0..n-2]$

- ⌚ Decrease-by-one-and-Conquer technique
- ⌚ Usually implemented bottom up (non-recursively)

Example:

89		<b>45</b>	68	90	29	34	17
45	89		<b>68</b>	90	29	34	17
45	68	89		<b>90</b>	29	34	17
45	68	89	90		<b>29</b>	34	17
29	45	68	89	90		<b>34</b>	17
29	34	45	68	89	90		<b>17</b>
17	29	34	45	68	89	90	

If  $A[n - 1]$  elements are sorted, all we need to do is scan the sorted subarray from right to left until the

## 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**

$A[j + 1] \leftarrow A[j]$

$j \leftarrow j - 1$

$A[j + 1] \leftarrow v$

The basic operation of the algorithm is the key comparison  $A[j] > v$ . (Why not  $j \geq 0$ ?)

8

Why not  $j \geq 0$ ? Because it is almost certainly faster than the former in an actual computer implementation.

Moreover, it is not relevant to the algorithm it is just avoiding an out of bounds error.

a better implementation with a **sentinel**—see Problem 8 in this section's exercises—eliminates it altogether.

- In general, a sentinel value makes it possible to detect the end of the data when no out-of-band data is available.
- In general the value should be selected in such a way that it is guaranteed to be distinct from all legitimate data values.
- In this case, the sentinel should stop the smallest element from moving beyond the first position in the array.

# Analysis of Insertion Sort

## ⌚ Time efficiency

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

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

$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2).$$

⌚ Space efficiency: in-place or  $O(1)$  or constant

⌚ Stability: yes

⌚ Best elementary sorting algorithm overall

- Binary insertion sort = insertion sort + binary search

9

The worst-case input is an array of strictly decreasing values.

Thus, in the worst case, insertion sort makes exactly the same number of comparisons as selection sort.

The best case is when the array is already sorted.

- Quick sort is worst on an already sorted array.

A rigorous analysis of the algorithm's average-case efficiency is based on investigating the number of