CPSC 5031 : Data Structures & Algorithms

Lecture 4: Decrease-and-Conquer

(Not in Sedgewick, but a common pattern)

Decrease-and-Conquer

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

Decrease-(by one)-and-conquer

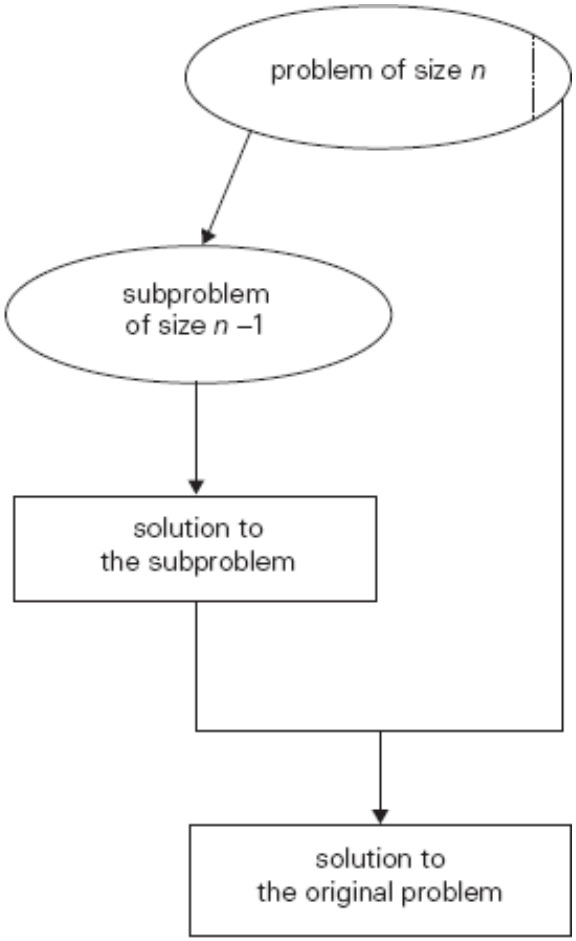

FIGURE 4.1 Decrease-(by one)-and-conquer technique.

Decrease-(by half)-and-conquer

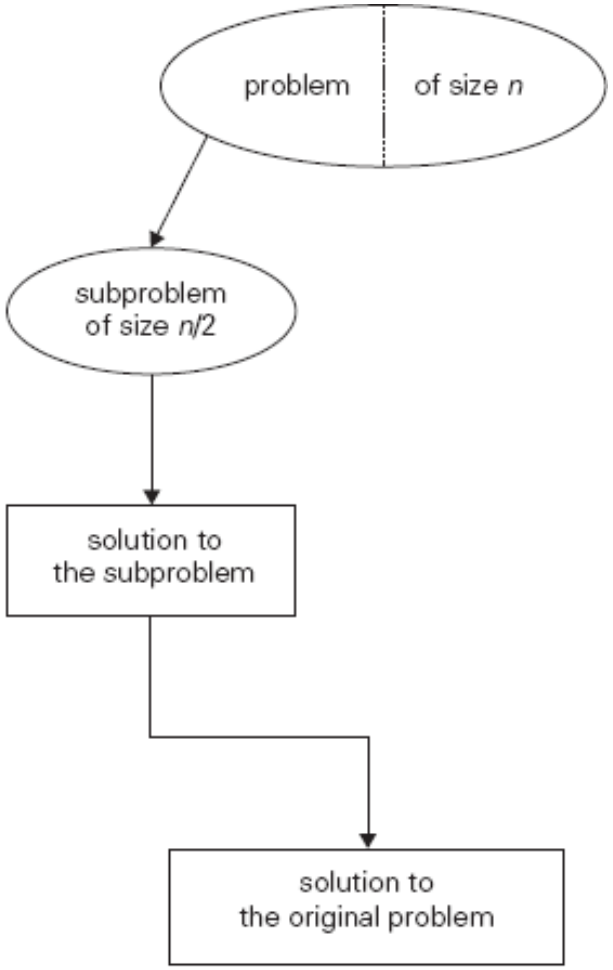

FIGURE 4.2 Decrease-(by half)-and-conquer technique.

Problems to Discuss

- Decrease-by-a-constant
  - Exponentiation: compute a^n
  - Insertion sorting
- Decrease-by constant-factor
  - Revisit exponentiation: compute a^n
  - Binary Search
  - Fake-Coin
- Variable-Size-Decrease
  - Save for Divide & Conquer lecture

Problems to Discuss

- **Decrease-by-a-constant**
  - Exponentiation: compute a^n
  - Insertion sorting
- Decrease-by constant-factor
  - Revisit exponentiation: compute a^n
  - Binary Search
  - Fake-Coin
- Variable-Size-Decrease
  - Save for Divide & Conquer lecture

Compute a^n

- May be computed either "top down" by using its recursive definition or "bottom up" by multiplying 1 by *a n* times.
- Same as brute-force algorithm.

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

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]

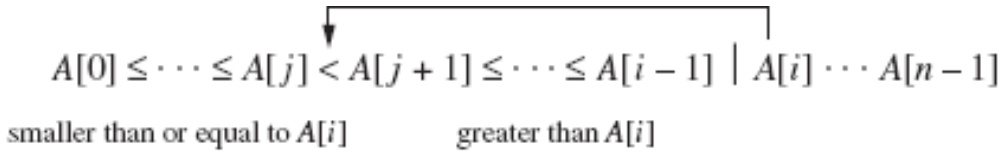

FIGURE 4.3 Iteration of insertion sort: $A[i]$ is inserted in its proper position among the preceding elements previously sorted.

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

FIGURE 4.4 Example of sorting with insertion sort. A vertical bar separates the sorted part of the array from the remaining elements; the element being inserted is in bold.

Pseudocode for 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 ← 1 to n − 1 do
    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
```

A different look at Insertion Sort

- Apply decrease-and-conquer!

- Time efficiency

  $C_{worst}(n) = n(n\text{-}1)/2 \in \Theta(n^2)$

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

  $C_{best}(n) = n - 1 \in \Theta(n)$ (also fast on almost sorted arrays)

Generating Subsets

- Problem: generating all 2^n subsets of set A={$a_1, \ldots, a_n$}
- Discussion: how?

Generating Subsets

- Problem: generating all 2^n subsets of set A={a_1, …, a_n}
- Solution 1: decrease by one and conquer
  - Generating all 2^{n-1} subsets for {a_1, …, a_n}
  - Add a_n to each subset

n	subsets							
0	∅							
1	∅	$\{a_1\}$						
2	∅	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$				
3	∅	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$	$\{a_3\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$

FIGURE 4.10 Generating subsets bottom up.

Generating Subsets

- Problem: generating all 2^n subsets of set A={$a_1, \ldots, a_n$}
- Solution 1: decrease by one and conquer
  - Generating all 2^{n-1} subsets for {$a_1, \ldots, a_n$}
  - Add a_n to each subset
- Solution 2: Using a bit string of length n
  - Example: 000, 001, …,111

Problems to Discuss

- Decrease-by-a-constant
  - Exponentiation: compute a^n
  - Insertion sorting
- **Decrease-by constant-factor**
  - Revisit exponentiation: compute a^n
  - Binary Search
  - Fake-Coin
- Variable-Size-Decrease
  - Save for Divide & Conquer lecture

Compute a^n

A decrease-by-a-constant-factor technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm

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

Binary Search

Very efficient algorithm for searching in sorted array:

K

vs

A[0] . . . A[m] . . . A[n-1]

If K = A[m], stop (successful search); otherwise, continue searching by the same method in A[0..m-1] if K < A[m] and in A[m+1..n-1] if K > A[m]

$l \leftarrow 0$; $r \leftarrow n-1$
while $l \leq r$ do
    $m \leftarrow \lfloor (l+r)/2 \rfloor$
    if $K = A[m]$ return m
    else if $K < A[m]$ $r \leftarrow m-1$
    else $l \leftarrow m+1$
return -1

Analysis of Binary Search

- Time efficiency
  - worst-case recurrence: $C_w(n) = 1 + C_w(\lfloor n/2 \rfloor)$, $C_w(1) = 1$
    solution: $C_w(n) = \lceil \log_2(n+1) \rceil$

    This is VERY fast: e.g., $C_w(10^6) = 20$

- Optimal for searching a sorted array

- Limitations: must be a sorted **array (not linked list)**

- Bad (degenerate) example of divide-and-conquer
  - (we'll talk about this in the next lecture)

Aside: choosing the right data structure

- Limitations: must be a sorted **array (not linked list)**

Data structures have a few fundamental operations:

- Create structure
- Insert element
- Fetch a given element
- Search for an element
- Delete an element
- Delete structure

Each has its own O()!

Linked Lists

- Limitations: must be a sorted **array (not linked list)**

Data structures have a few fundamental operations:

- Create structure – O(1)
- Insert element – O(n)
- Fetch a given element – O(n)
- Search for an element – O(n)
- Delete an element – O(1)
- Delete structure – O(n)

Each has its own O()!

Arrays

- Limitations: must be a sorted **array (not linked list)**

Data structures have a few fundamental operations:

- Create structure – O(1)
- Insert element – O(n) (copying the old array!)
- Fetch a given element – O(1)
- Search for an element – O(log n) if sorted, otherwise O(n)
- Delete an element – O(1) with sentinels, otherwise O(n)
- Delete structure – O(1)

Each has its own O()!

Fake Coin Puzzle

There are *n* identically looking coins one of which is fake. There is a balance scale but there are no weights; the scale can tell whether two sets of coins weigh the same and, if not, which of the two sets is heavier (but not by how much).

Design an efficient algorithm for detecting the fake coin. Assume that the fake coin is known to be lighter than the genuine ones.

- How would you solve this problem by applying decrease-and-conquer?

Fake Coin Puzzle

- Solution 1: Decrease by factor 2 algorithm
  - —Divide n coins into two piles of $\lfloor n/2 \rfloor$ coins each, leaving one extra coin aside if n is odd
  - – Put piles on scale
    - If piles are same, coin set aside must be fake
    - Else, proceed in same manner with lighter pile

$W(n) = W(\lfloor n/2 \rfloor) + 1$ for $n > 1$, $W(1) = 0$.