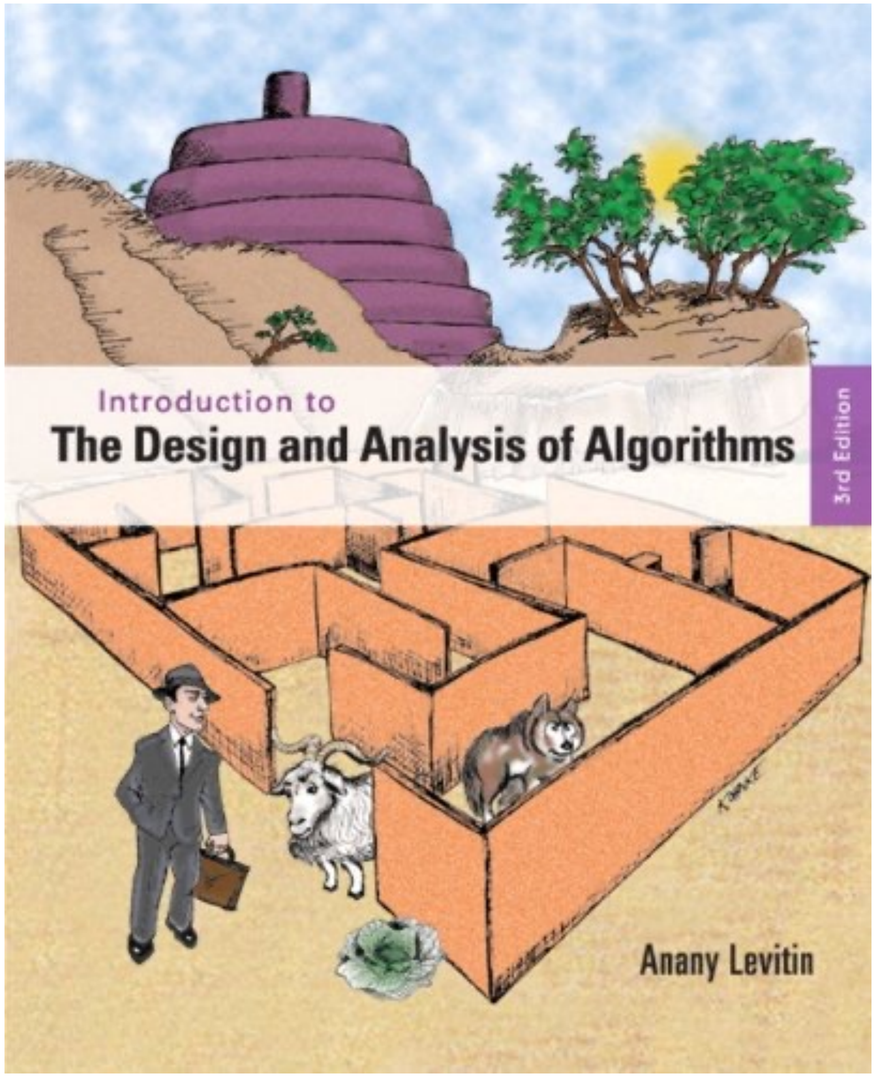
CSC 411
Design and Analysis of Algorithms

Chapter 4 Decrease-and-Conquer

Instructor: Lin-Ching Chang

Chapter 4

Decrease-and-Conquer



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

- Can be implemented either top-down or bottom-up
- Also referred to as *inductive* or *incremental* approach

3 Types of Decrease and Conquer

- *Decrease by a constant* (usually by 1):
  - insertion sort
  - graph traversal algorithms (DFS and BFS)
  - 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

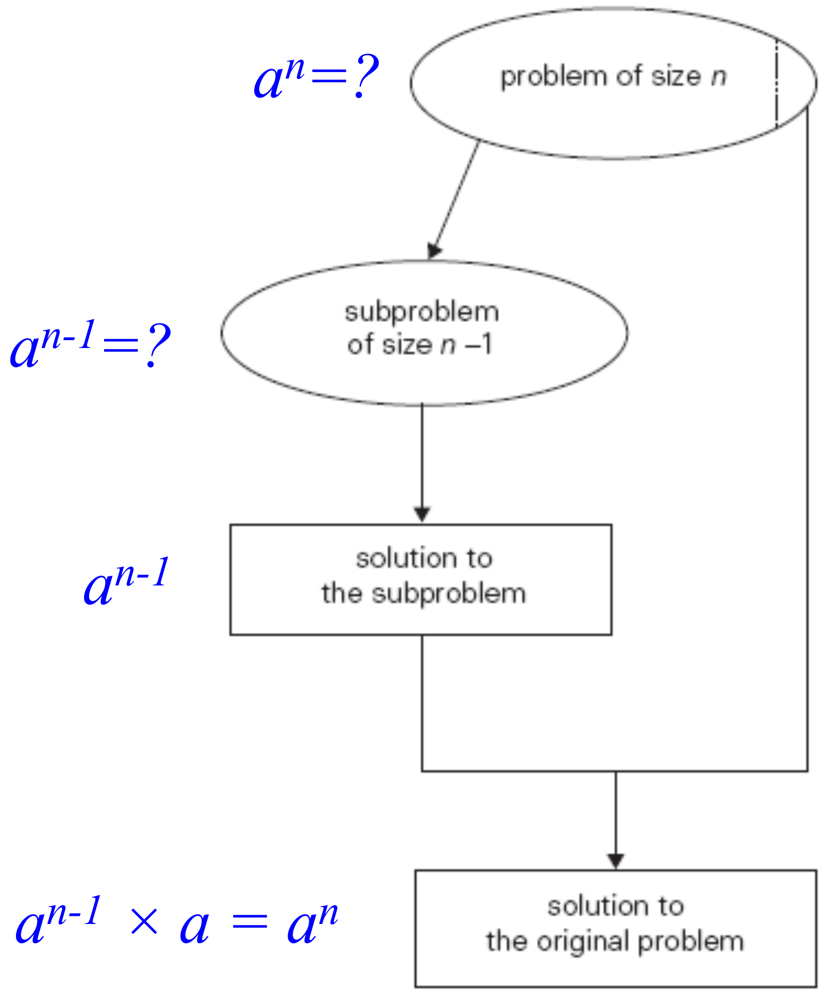
What’s the difference?

Consider the problem of exponentiation: Compute a^n

- Brute Force:
- Divide and Conquer:
- Decrease by one:
- Decrease by constant factor:

What is the concept of each design strategy?

Decrease by a constant (usually by 1)



$a^n=?$
problem of size *n*
$a^{n-1}=?$
subproblem of size *n* –1
a^{n-1}
solution to the subproblem
$a^{n-1} \times a = a^n$
solution to the original problem

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

Decrease by a constant factor (usually by half)

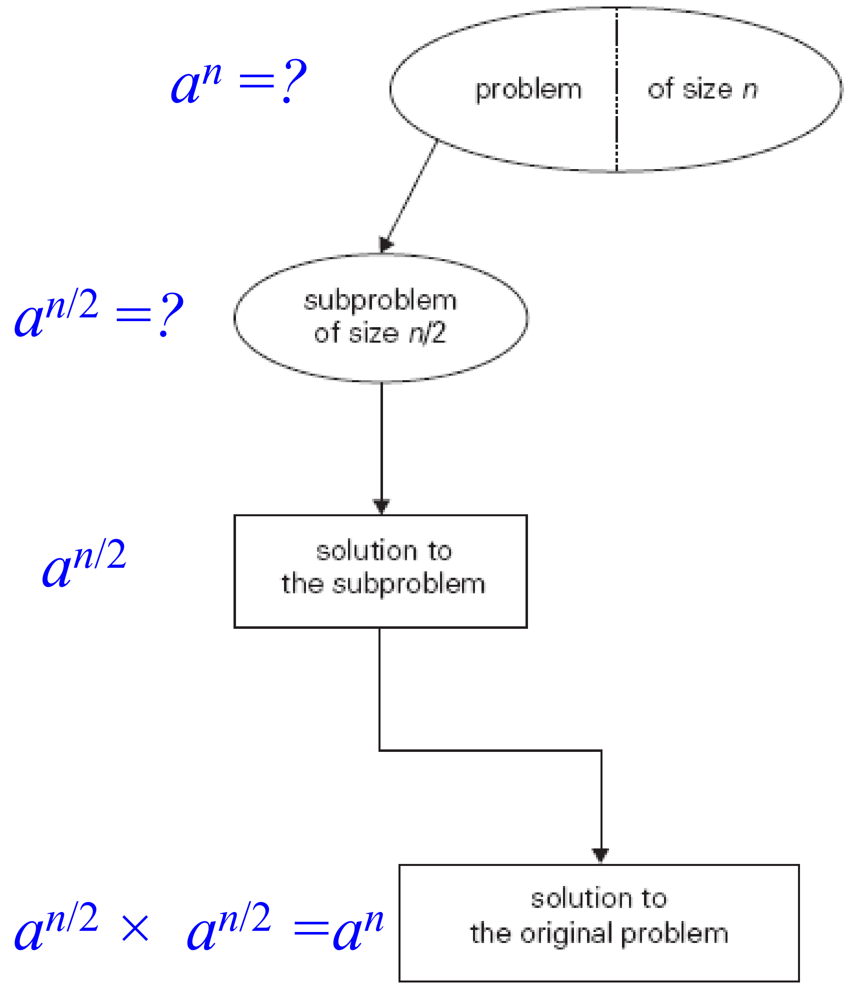
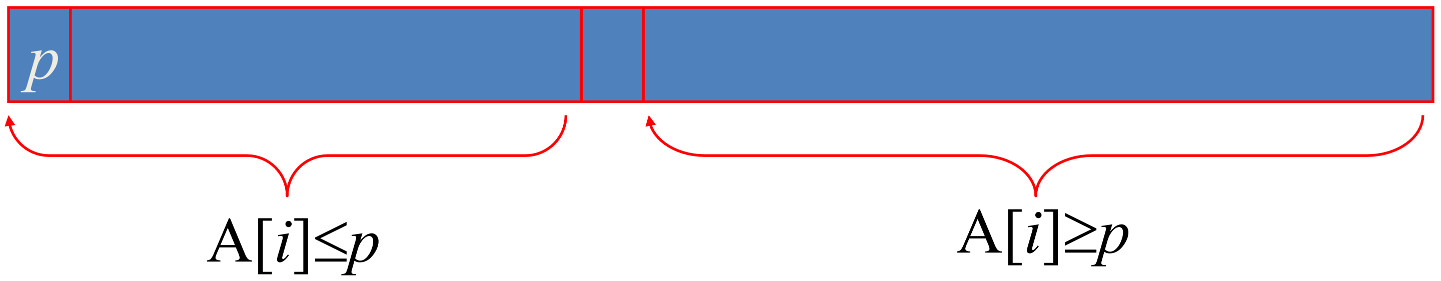


FIGURE 4.2 Decrease-(by half)-and-conquer technique.

Variable-size decrease

- Euclid's algorithm
  - gcd(m,n) = gcd(n, m mod n)
- Quick sort



What's the difference?

Consider the problem of exponentiation: Compute a^n

- Brute Force:
  - $a^n = a \times a \times a \times a \times \ldots \times a$
- Divide and Conquer:
  - $a^n = a^{n/2} \times a^{n/2}$ (more accurately, $a^n = a^{\lfloor n/2 \rfloor} \times a^{\lceil n/2 \rceil}$)
- Decrease by one:
  - $a^n = a^{n-1} \times a$
  - $f(n) = f(n-1) \times a$ if $n > 1$, and $f(1) = a$
- Decrease by constant factor:
  - $a^n = (a^{n/2})^2$ if n is even
  - $a^n = (a^{(n-1)/2})^2 \times a$ if n is odd

Master Theorem:

If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

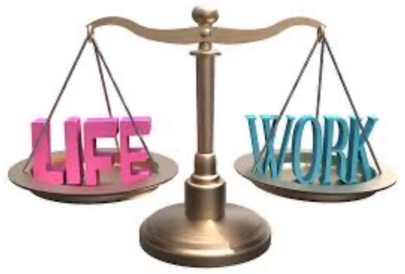
If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

More of this design strategy will be explained later in chapter 5.

Fake-Coin Puzzle (simpler version)

- 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.

- Brute Force:
- Divide and conquer:
- Decrease by constant factor:



9 Coins Puzzle

There are 9 identically looking coins one of which is fake. However, we don't know the fake coin is lighter or heavier. 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 and to tell the fake coin is light or heavier.

Decrease by factor 2 algorithm?

Decrease by factor 3 algorithm?

Fake-Coin Puzzle (n coins, one fake)

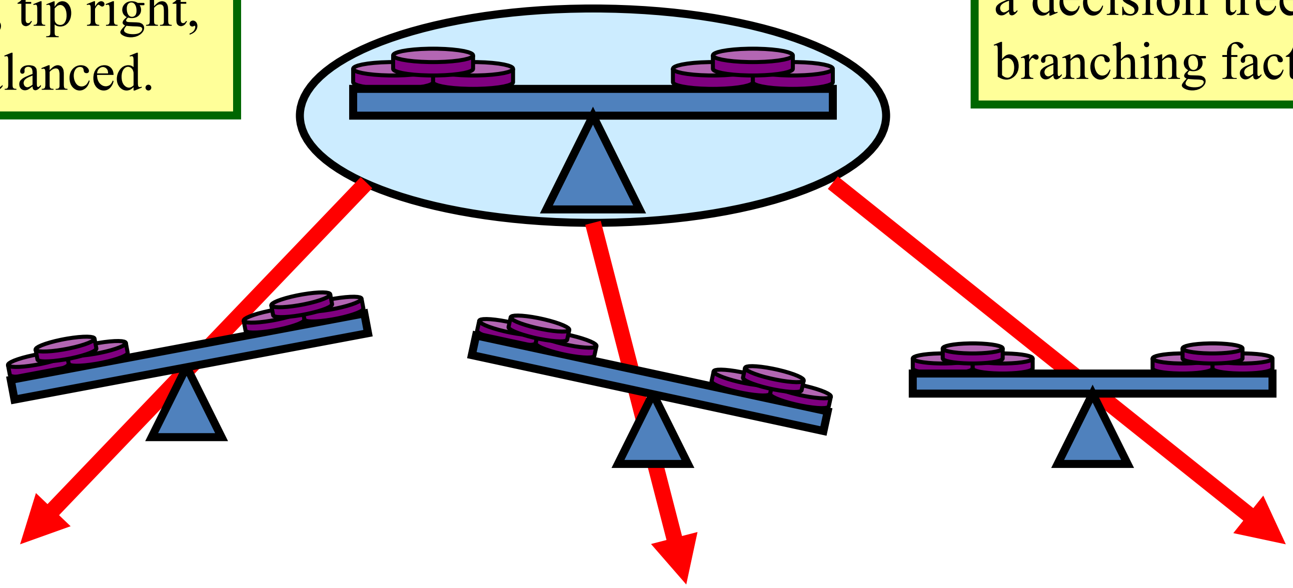
- Assume that the fake coin is known to be lighter than the genuine ones:
  - Decrease by factor 2 algorithm -> $O(\log_2 n)$
  - Decrease by factor 3 algorithm -> $O(\log_3 n)$
    - We need w weighing with a balance to find a light counterfeit coin among 3^w coins.
    - So, the number of required weighings with n coins is $w = \lceil \log_3 n \rceil$.

- It is a Decision-Tree Problem
  - In each situation, we pick two disjoint and equal-size subsets of coins to put on the scale.

The balance then "decides" whether to tip left, tip right, or stay balanced.

Each weighing has 3 possible outcomes

A given sequence of weighings thus yields a decision tree with branching factor 3.



Tree Height Theorem

- How many weighings will be necessary?
  - If you have 8 coins?
    - The decision tree must have at least 8 leaf nodes, since there are 8 possible outcomes.
  - If you have 9 coins?
    - The decision tree must have at least 9 leaf nodes, since there are 9 possible outcomes.
  - If you have 12 coins?
    - The decision tree must have at least 12 leaf nodes, since there are 12 possible outcomes.

Tree Height Theorem

- There are at most m^h leaves in an *m*-ary tree of height *h*.
  - **Corollary:** An *m*-ary tree with ℓ leaves has height $h \geq \lceil \log_m \ell \rceil$. If *m* is full and balanced then $h = \lceil \log_m \ell \rceil$.

- What is this decision tree's height?
  - Binary tree
  - If you have 8 leaves? Height = 3
  - If you have 9 leaves? Height = 4
  - If you have 12 leaves? Height = 4

- What is this decision tree's height?
  - 3-ary tree
  - If you have 8 leaves? Height = 2
  - If you have 9 leaves? Height = 2
  - If you have 12 leaves? Height = 3

General Balance Strategy

- On each step, put $\lceil n/3 \rceil$ of the n coins to be searched on each side of the scale.
  - If the scale tips to the left, then:
    - The lightweight fake is in the right set of $\lceil n/3 \rceil \approx n/3$ coins.
  - If the scale tips to the right, then:
    - The lightweight fake is in the left set of $\lceil n/3 \rceil \approx n/3$ coins.
  - If the scale stays balanced, then:
    - The fake is in the remaining set of $n - 2\lceil n/3 \rceil \approx n/3$ coins that were not weighed!
- Except if $n \bmod 3 = 1$ then we can do a little better by weighing $\lfloor n/3 \rfloor$ of the coins on each side.

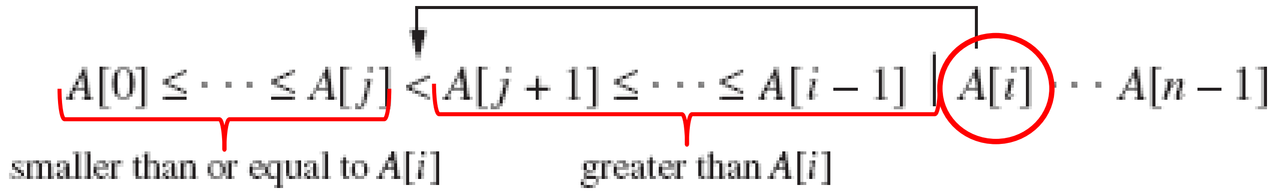
You can prove that this strategy always leads to a balanced 3-ary tree.

- How should we change the procedures above if the fake coin is known to be heavier than normal ones?

- How should we change the procedures above if we don’t know the fake coin is lighter or heavier ?

4.1 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]
- Usually implemented bottom up (nonrecursively)



$A[0] \le \cdots \le A[j] < A[j+1] \le \cdots \le A[i-1] \mid A[i] \cdots A[n-1]$
smaller than or equal to $A[i]$
greater than $A[i]$

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

Example: Sort 6, 4, 1, 8, 5

6 | $\underline{4}$ 1 8 5
4 6 | $\underline{1}$ 8 5
1 4 6 | $\underline{8}$ 5
1 4 6 8 | $\underline{5}$
1 4 5 6 8

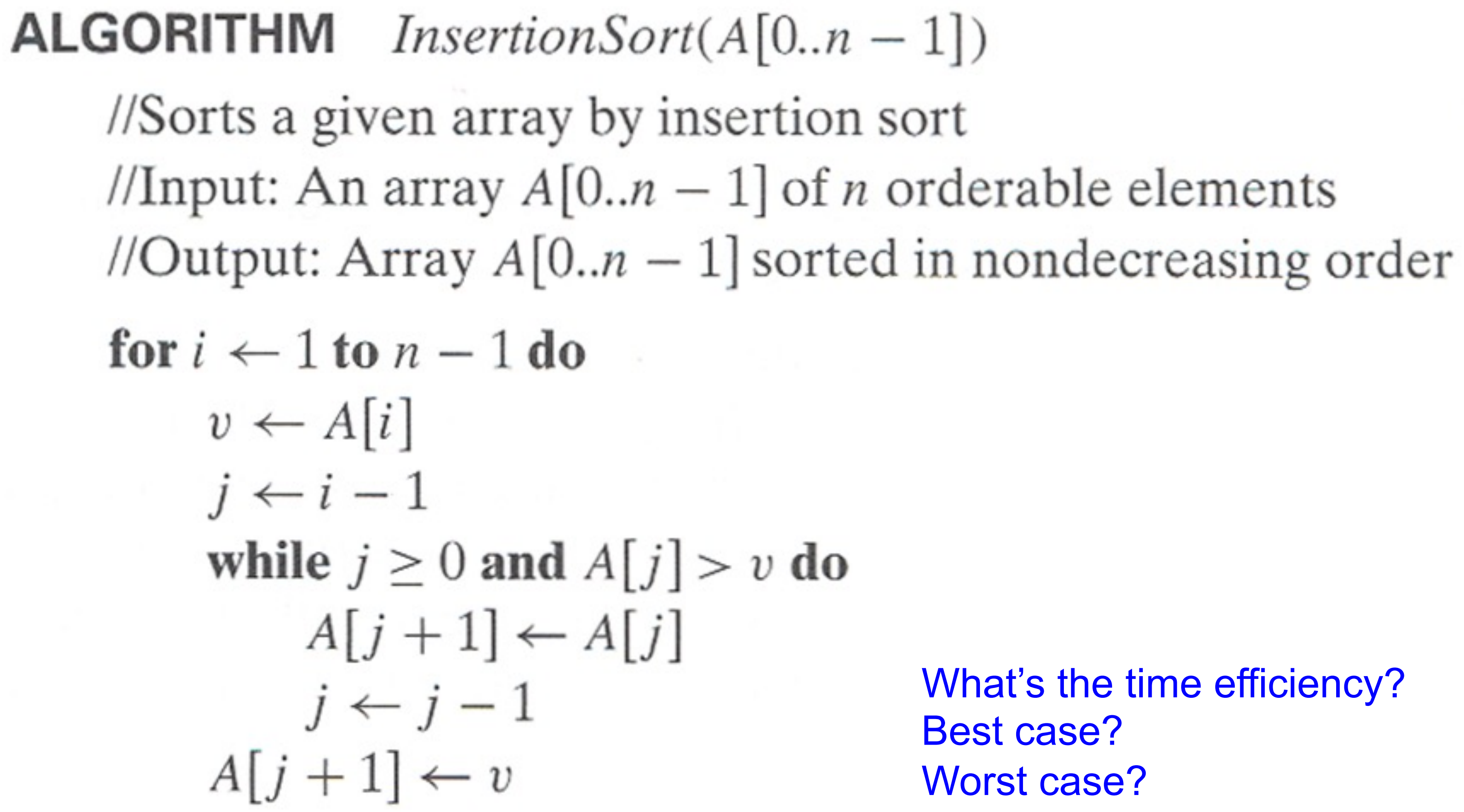
Insertion Sort

Example: Sort 89, 45, 68, 90, 25, 34, 17 (total n=7)

Index 0	1	2	3,	4	5	6
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

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 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
            $A[j+1] \leftarrow A[j]$
            $j \leftarrow j-1$
        $A[j+1] \leftarrow v$
```

What's the time efficiency?
Best case?
Worst case?

Analysis of Insertion Sort

- Time efficiency ?

$C_{worst}(n) = n(n\text{-}1)/2 \in \Theta(n^2)$ when array is already sorted in reversed order

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

$C_{best}(n) = n - 1 \in \Theta(n)$ when array is already sorted

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

$C_{avg}(n) \approx$?

$\approx n^2/4 \in \Theta(n^2)$