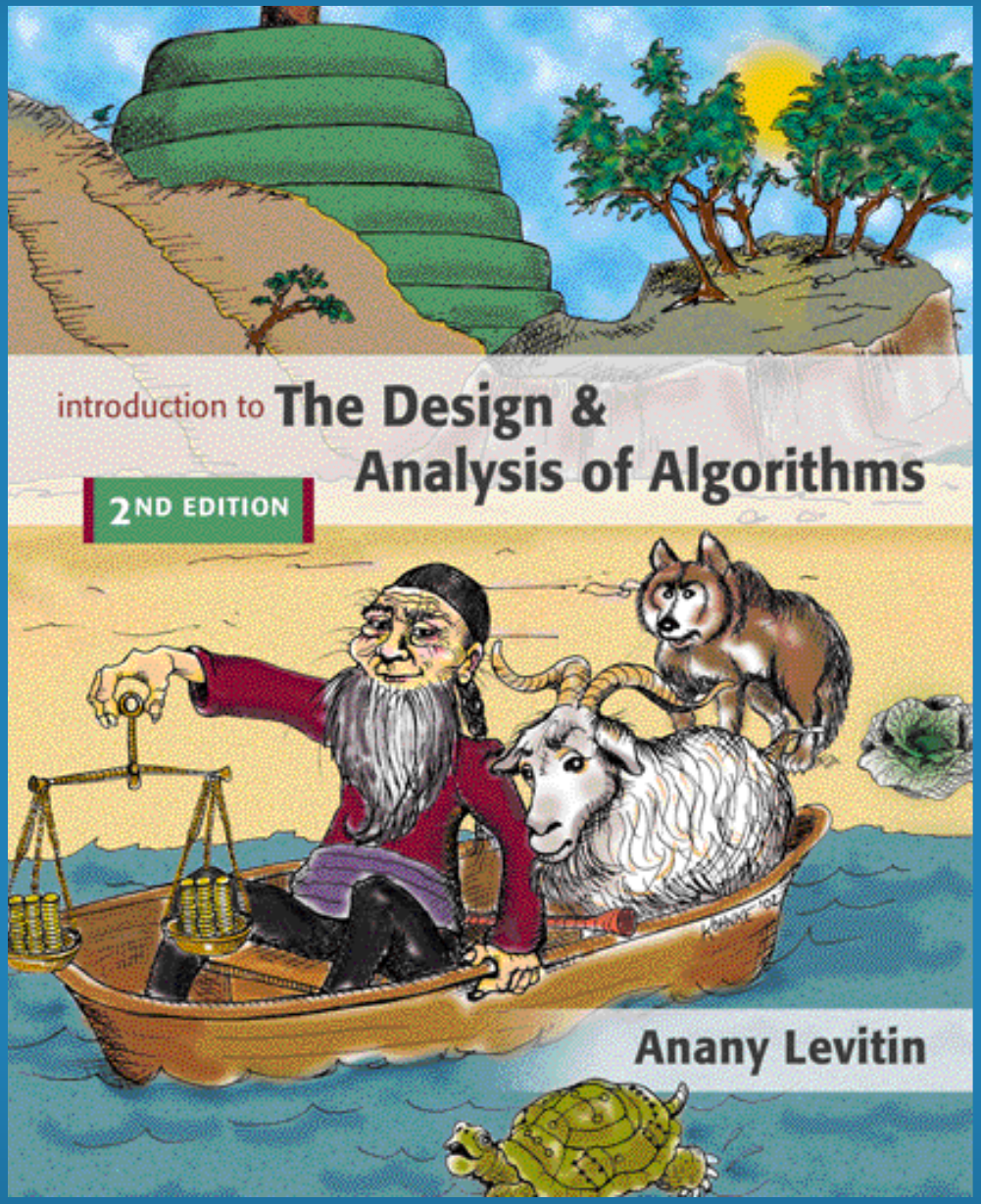
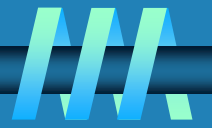
Decrease-and-Conquer

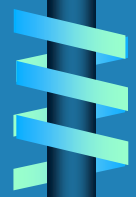


Decrease-and-Conquer

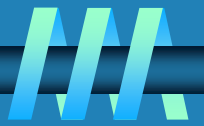


1. Reduce problem instance to smaller instance of the same problem
2. Solve smaller instance
3. Extend solution of smaller instance to obtain solution to original instance

- 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**

- ***Variable-size decrease***
  - **Euclid's algorithm**

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)**

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

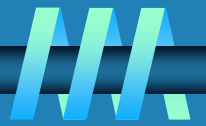
6 | 4 1 8 5
4 6 | 1 8 5
1 4 6 | 8 5
1 4 6 8 | 5
1 4 5 6 8

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

Analysis of Insertion Sort



- **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)

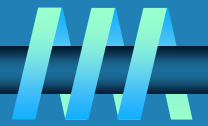
- **Space efficiency:**

  in-place

Stability:

yes

- **Best elementary sorting algorithm overall**

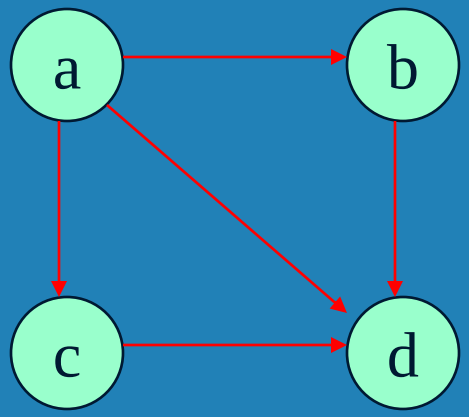


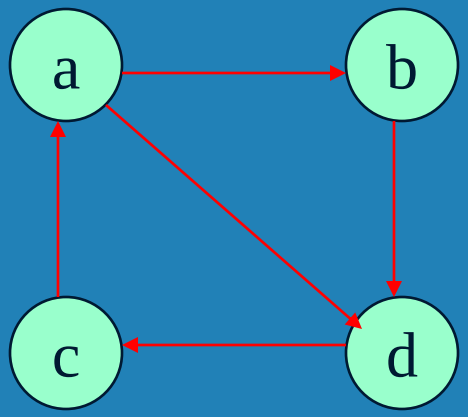
Ferrying soldiers A detachment of n soldiers must cross a wide and deep river with no bridge in sight. They notice two 12-year-old boys playing in a rowboat by the shore. The boat is so tiny, however, that it can only hold two boys or one soldier. How can the soldiers get across the river and leave the boys in joint possession of the boat? How many times need the boat pass from shore to shore?

DAGs and Topological Sorting

A *dag*: a directed acyclic graph, i.e. a directed graph with no (directed) cycles

a dag





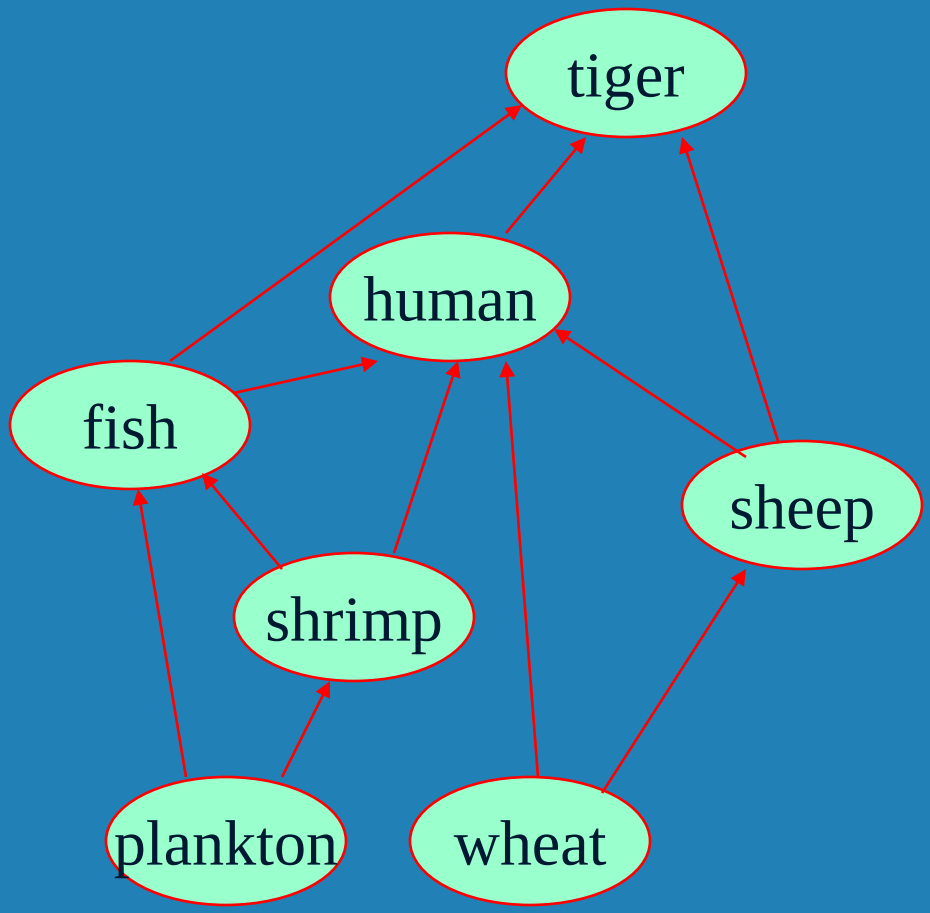
not a dag

Arise in modeling many problems that involve prerequisite constraints (construction projects, document version control)

Vertices of a dag can be linearly ordered so that for every edge its starting vertex is listed before its ending vertex (*topological sorting*). Being a dag is also a necessary condition for topological sorting to be possible.

Topological Sorting Example

Order the following items in a food chain

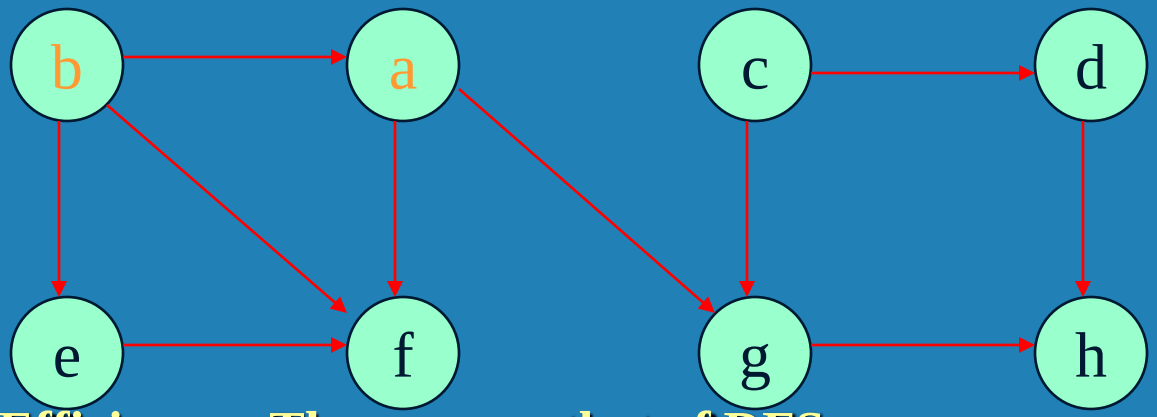


DFS-based Algorithm

DFS-based algorithm for topological sorting

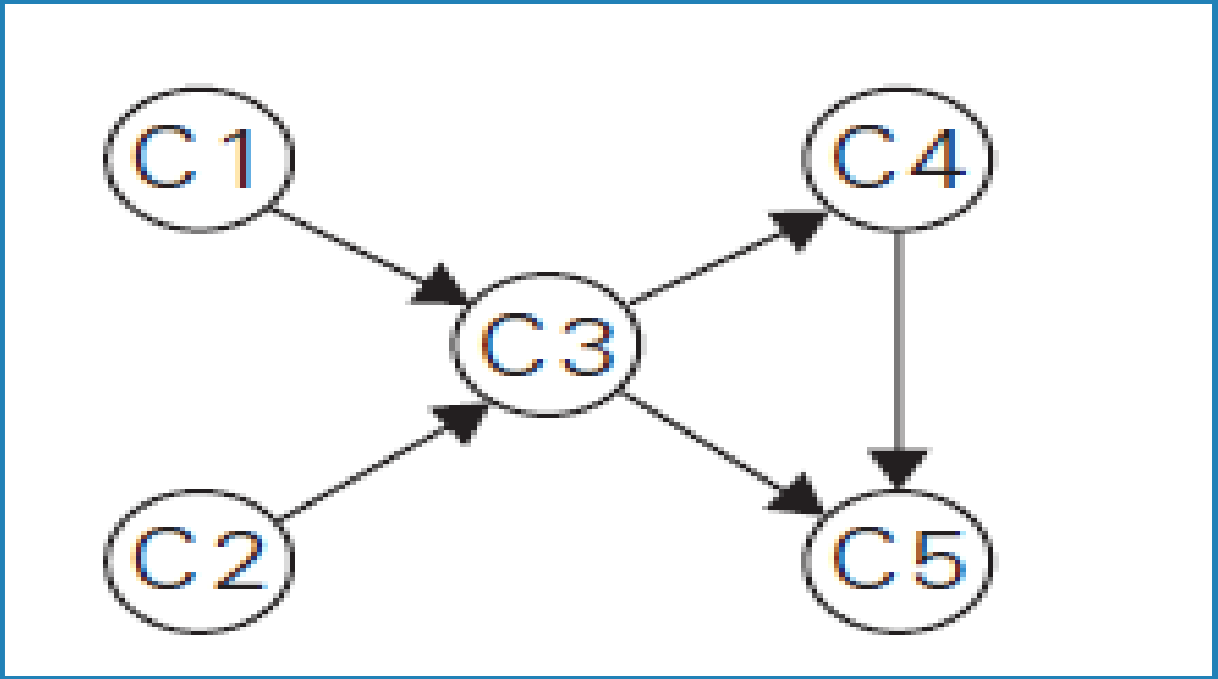
- **Perform DFS traversal, noting the order vertices are popped off the traversal stack**
- **Reverse order solves topological sorting problem**
- **Back edges encountered?→ NOT a dag!**

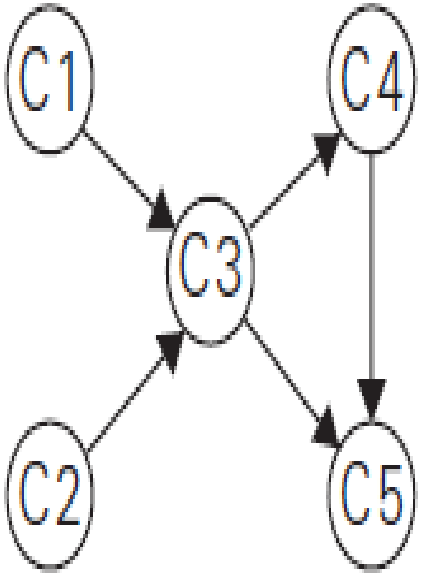
Example:



Efficiency: The same as that of DFS.

Example





$C5_1$

$C4_2$

$C3_3$

$C1_4$ $C2_5$

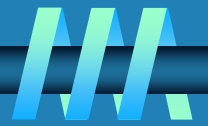
The popping-off order:

C5, C4, C3, C1, C2

The topologically sorted list:

C2 C1→C3→C4→C5

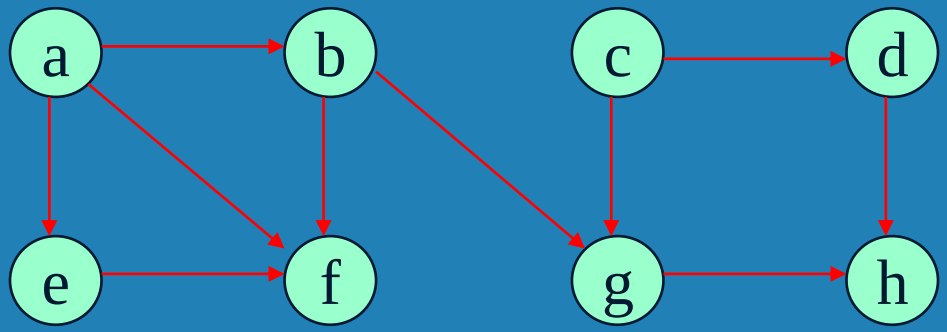
Source Removal Algorithm



Source removal algorithm

Repeatedly identify and remove a *source* (a vertex with no incoming edges) and all the edges incident to it until either no vertex is left or there is no source among the remaining vertices (not a dag)

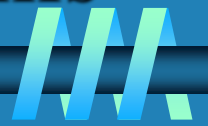
Example:



Efficiency: same as efficiency of the DFS-based algorithm, but how would you identify a source? How do you remove a source from the dag?

"Invert" the adjacency lists for each vertex to count the number of incoming edges by going thru each adjacency list and counting the number of times that each vertex appears in these lists. To remove a source, decrement the count of each of its neighbors by one.

Decrease-by-Constant-Factor Algorithms

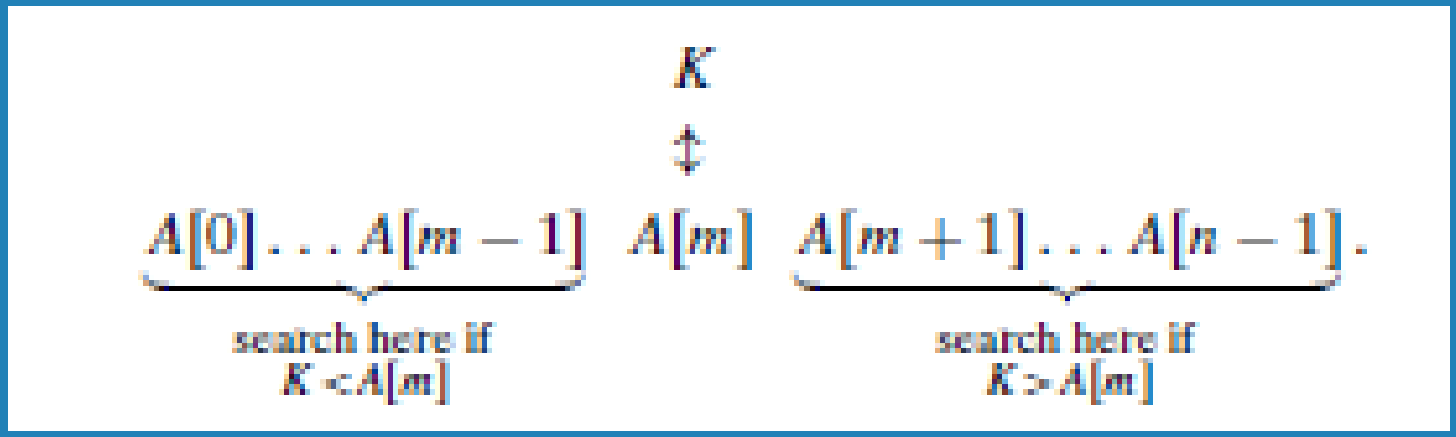


In this variation of decrease-and-conquer, instance size is reduced by the same factor (typically, 2)

Examples:

- **Binary search and**

Binary Search

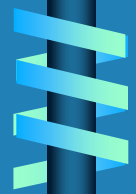


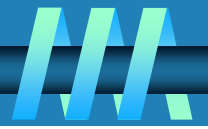
3	14	27	31	39	42	55	70	74	81	85	93	98

ALGORITHM *BinarySearch*$(A[0..n-1], K)$

```
//Implements nonrecursive binary search
//Input: An array A[0..n − 1] sorted in ascending order and
//       a search key K
//Output: An index of the array's element that is equal to K
//       or −1 if there is no such element
l ← 0;   r ← n − 1
while l ≤ r do
    m ← ⌊(l + r)/2⌋
    if K = A[m] return m
    else if K < A[m] r ← m − 1
    else l ← m + 1
return −1
```

$$C_{worst}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n+1) \rceil.$$





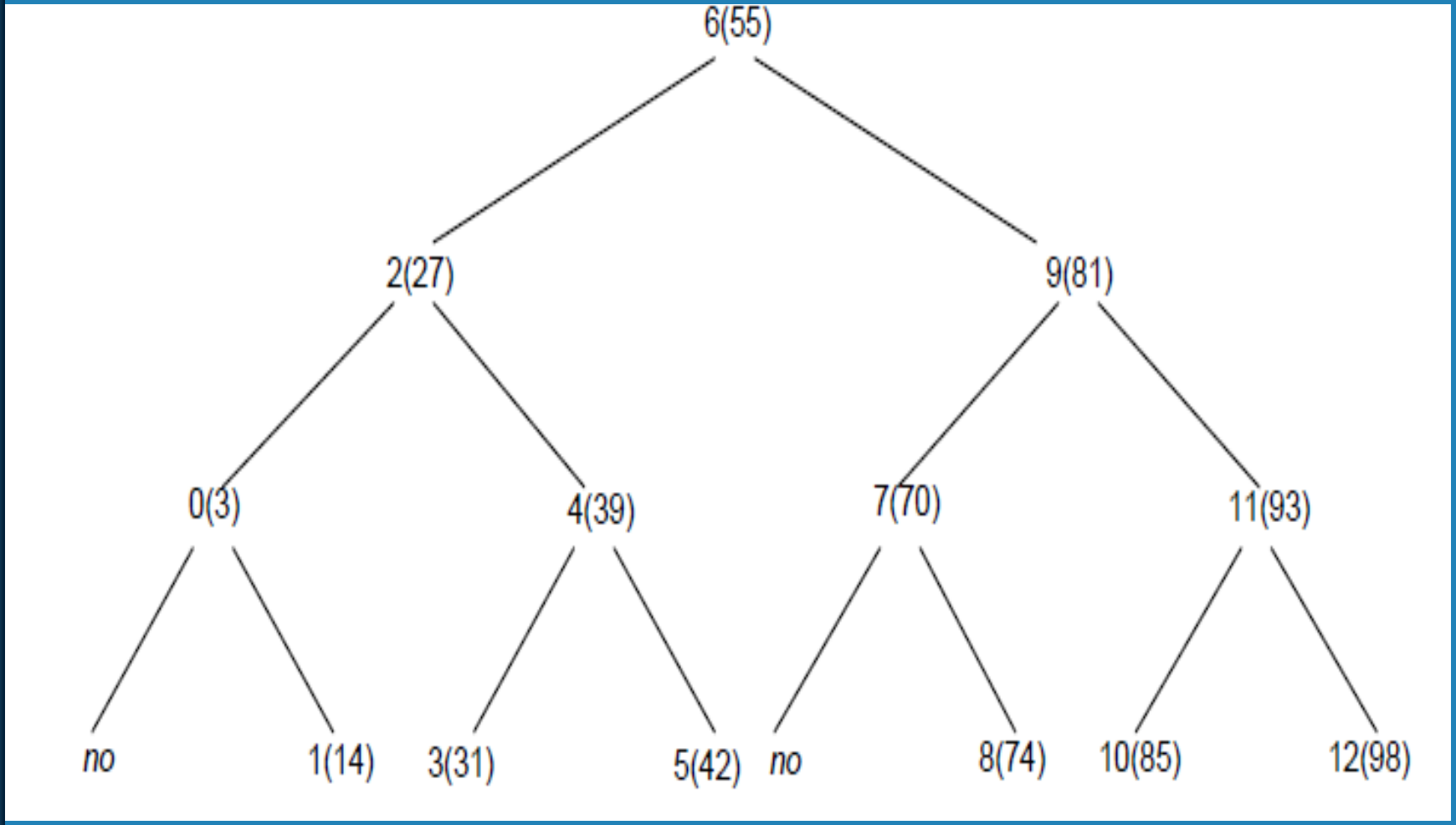
What is the largest number of key comparisons made by binary search in searching for a key in the following array?

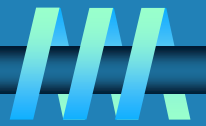
3	14	27	31	39	42	55	70	74	81	85	93	98

List all the keys of this array that will require the largest number of key comparisons when searched for by binary search.

Find the average number of key comparisons made by binary search in a successful search in this array. Assume that each key is searched for with the same probability.

Find the average number of key comparisons made by binary search in an unsuccessful search in this array. Assume that searches for keys in each of the 14 intervals formed by the array's elements are equally likely.

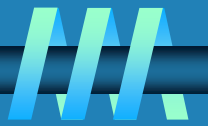




c. $C_{avg}^{yes} = \frac{1}{13} \cdot 1 \cdot 1 + \frac{1}{13} \cdot 2 \cdot 2 + \frac{1}{13} \cdot 3 \cdot 4 + \frac{1}{13} \cdot 4 \cdot 6 = \frac{41}{13} \approx 3.2.$

d. $C_{avg}^{no} = \frac{1}{14} \cdot 3 \cdot 2 + \frac{1}{14} \cdot 4 \cdot 12 = \frac{54}{14} \approx 3.9.$

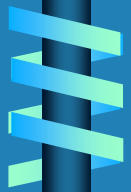
Variable-Size-Decrease Algorithms



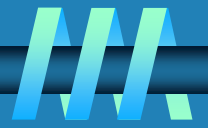
In the variable-size-decrease variation of decrease-and-conquer, instance size reduction varies from one iteration to another

Examples:

- **Euclid's algorithm for greatest common divisor**



Euclid's Algorithm



Euclid's algorithm is based on repeated application of equality

$$\gcd(m, n) = \gcd(n, m \bmod n)$$

Ex.: $\gcd(80,44) = \gcd(44,36) = \gcd(36, 8) = \gcd(8,4) = \gcd(4,0) = 4$