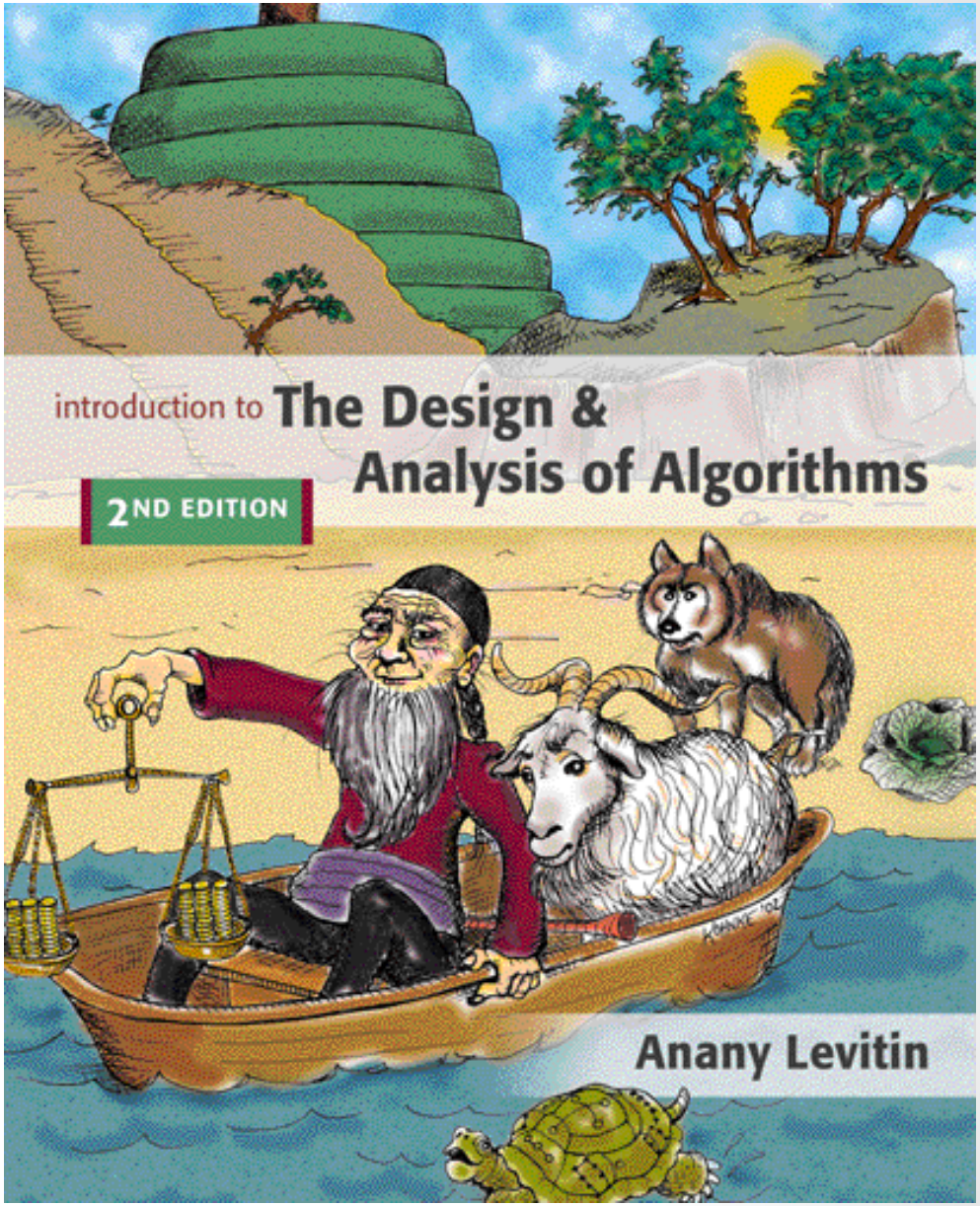
Chapter 5

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 (recursively) or bottom-up (without a recursion)
- Also referred to as *inductive* or *incremental* approach

3 Types of Decrease and Conquer

- *Decrease by a constant* (mesma a cada iteração, usually by 1):
  - insertion sort
  - graph traversal algorithms (DFS and BFS)
  - topological sorting
  - algorithms for generating permutations, subsets

- *Decrease by a constant factor* (mesmo a cada iteração, usually by half)
  - binary search and bisection method
  - exponentiation by squaring
  - multiplication *à la russe*

- *Variable-size decrease* (diferente a cada iteração)
  - Euclid's algorithm
  - selection by partition
  - Nim-like games

What's the difference?

Consider the problem of exponentiation: Compute a^n

- Brute Force:
  - a^n = a * a * a * a (n vezes) O(?)
- Decrease by one:
  - f(n) = { f(n-1)*a se i>=1, a se n=1} O(?)
- Divide and conquer:
  - $a^n = a^{\lfloor n/2 \rfloor} * a^{\lceil n/2 \rceil}$ se n >1 O(?)
  - a^n = a se n=1
- Decrease by constant factor: O(?)
  - $a^n = (a^{n/2})^2$ se n par e positivo
  - $a^n = (a^{(n-1)/2})^2 * a$ se n ímpar e >1
  - a^n = a se n=1

What's the difference?

Consider the problem of exponentiation: Compute a^n

- Brute Force:
  - a^n = a * a * a * a (n vezes) — O(n)
- Decrease by one:
  - f(n) = { f(n-1)*a se i>=1, a se n=1} — O(n)
- Divide and conquer:
  - $a^n = a^{\lfloor n/2 \rfloor} * a^{\lceil n/2 \rceil}$ se n >1 — O(n log n)
  - a^n = a se n=1
- Decrease by constant factor:
  - $a^n = (a^{n/2})^2$ se n par e positivo — O(log n)
  - $a^n = (a^{(n-1)/2})^2 * a$ se n ímpar e >1
  - a^n = a se n=1

Decrease-by-one: Insertion Sort

To sort array A[0..*n*-1], insert A[*n*-1] in its proper place among the sorted subarray 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 ← 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
```

Analysis of Insertion Sort

- Time efficiency

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

  $C_{avg}(n) \approx n^2/4 \in \Theta(n^2)$ (em média, 1/2 das comparações do pior caso)

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

- Space efficiency: in-place

- Stability: yes

- Best elementary sorting algorithm overall

- Binary insertion sort…. Binary search +insertion sort

Graph Traversal

Many problems require processing all graph vertices (and edges) in systematic fashion

Graph traversal algorithms:

- Depth-first search (DFS)
- Breadth-first search (BFS)

Depth-First Search (DFS)

- Visits graph's vertices by always moving away from last visited vertex to unvisited one, backtracks if no adjacent unvisited vertex is available.

- Uses a stack
  - a vertex is pushed onto the stack when it is reached for the first time
  - a vertex is popped off the stack when it becomes a dead end, i.e., when there is no adjacent unvisited vertex

- "Redraws" graph in tree-like fashion

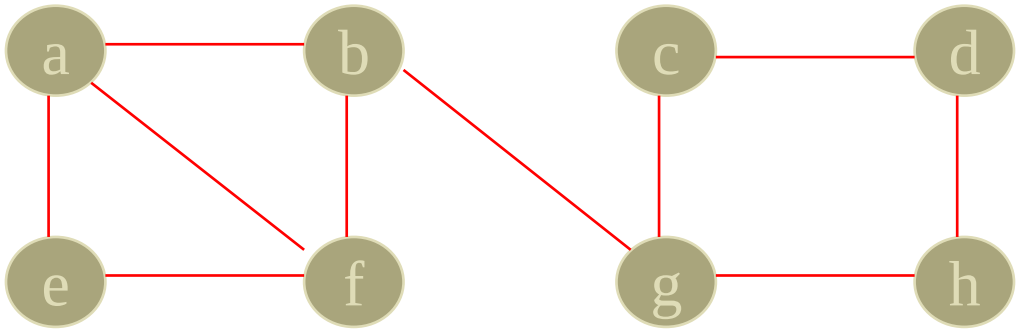
Pseudocode of DFS

ALGORITHM *DFS*(G)

//Implements a depth-first search traversal of a given graph
//Input: Graph $G = \langle V, E \rangle$
//Output: Graph G with its vertices marked with consecutive integers
//in the order they've been first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
$count \leftarrow 0$
for each vertex v in V **do**
    if v is marked with 0
        dfs(v)

dfs(v)
//visits recursively all the unvisited vertices connected to vertex v by a path
//and numbers them in the order they are encountered
//via global variable *count*
$count \leftarrow count + 1$; mark v with *count*
for each vertex w in V adjacent to v **do**
    if w is marked with 0
        dfs(w)

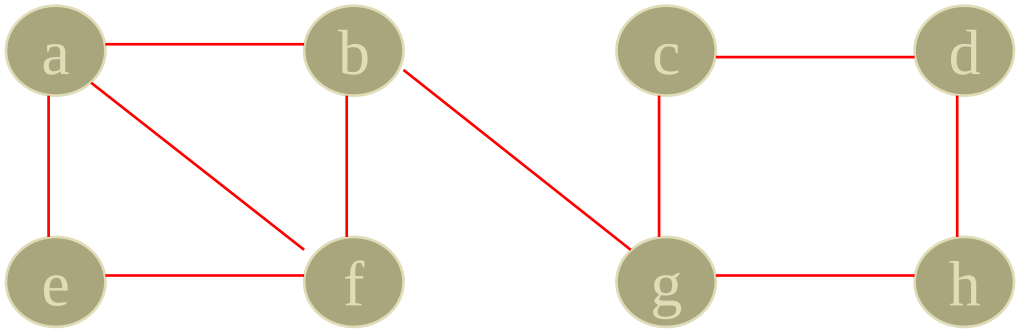
Example: DFS traversal of undirected graph



DFS traversal:

DFS tree:

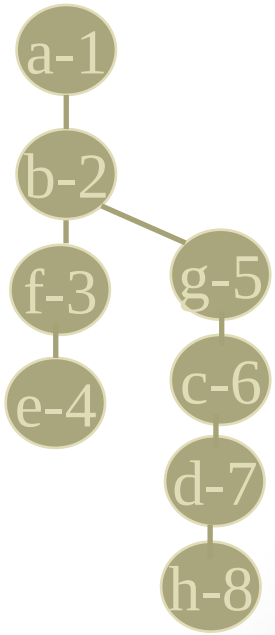
Example: DFS traversal of undirected graph



DFS traversal:

a b f e g c d h

DFS tree:



Notes on DFS

- DFS can be implemented with graphs represented as:
  - adjacency matrices: $\Theta(|V|^2)$
  - adjacency lists: $\Theta(|V|+|E|)$

- Yields two distinct ordering of vertices:
  - order in which vertices are first encountered (pushed onto stack)
  - order in which 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 (BFS)

- Visits graph vertices by moving across to all the neighbors of last visited vertex
- Instead of a stack, BFS uses a queue
- Similar to level-by-level tree traversal
- "Redraws" graph in tree-like fashion

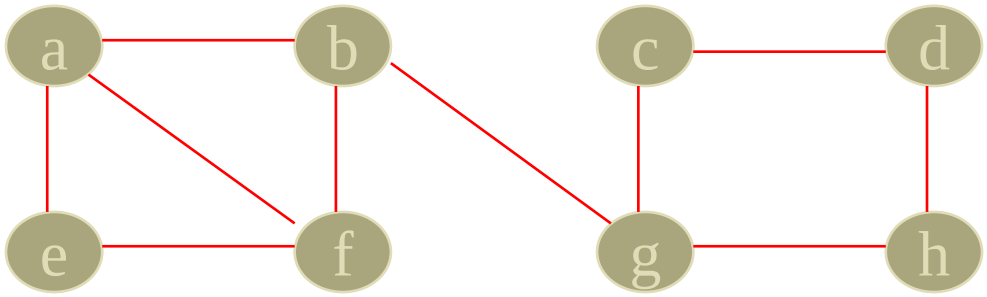
Pseudocode of BFS

ALGORITHM *BFS*(G)

```
//Implements a breadth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//in the order they have been visited by the BFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        bfs(v)

bfs(v)
//visits all the unvisited vertices connected to vertex v by a path
//and assigns them the numbers in the order they are visited
//via global variable count
count ← count + 1;  mark v with count and initialize a queue with v
while the queue is not empty do
    for each vertex w in V adjacent to the front vertex do
        if w is marked with 0
            count ← count + 1;  mark w with count
            add w to the queue
    remove the front vertex from the queue
```

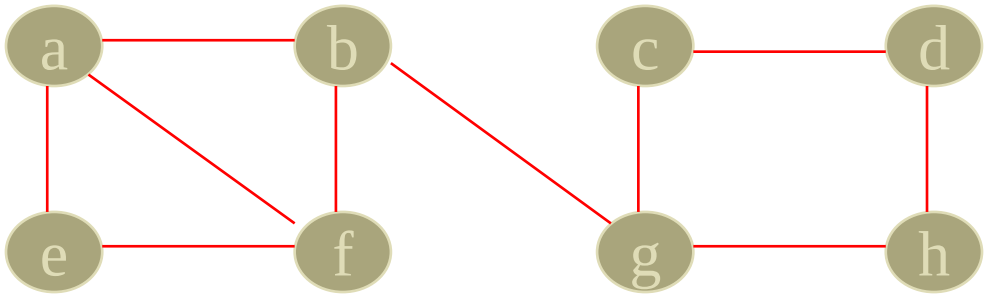
Example of BFS traversal of undirected graph



BFS traversal:

BFS tree:

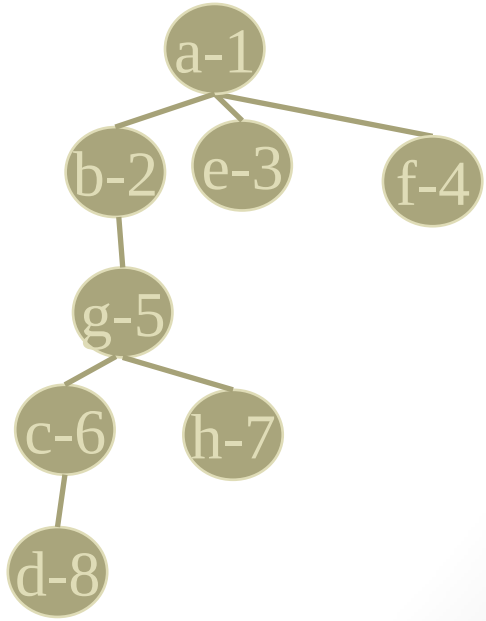
Example of BFS traversal of undirected graph



BFS traversal:*

a, b, e, f, g, c, h, d

BFS tree:



***depends on the adjacency lists order**

Notes on BFS

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

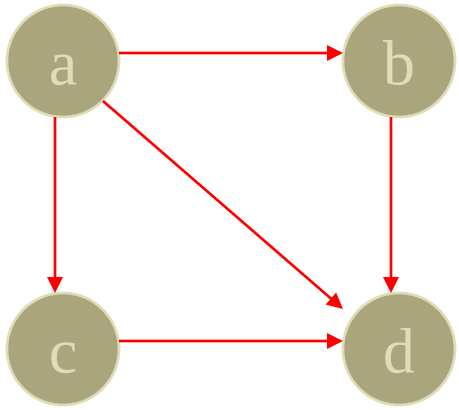
- Yields single ordering of vertices (order added/deleted from queue is the same)

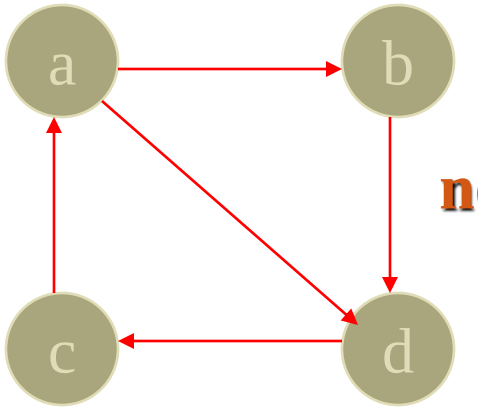
- Applications: same as DFS, but can also find paths from a vertex to all other vertices with the smallest number of edges (shortest path)

Dags and Topological Sorting

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

a dag





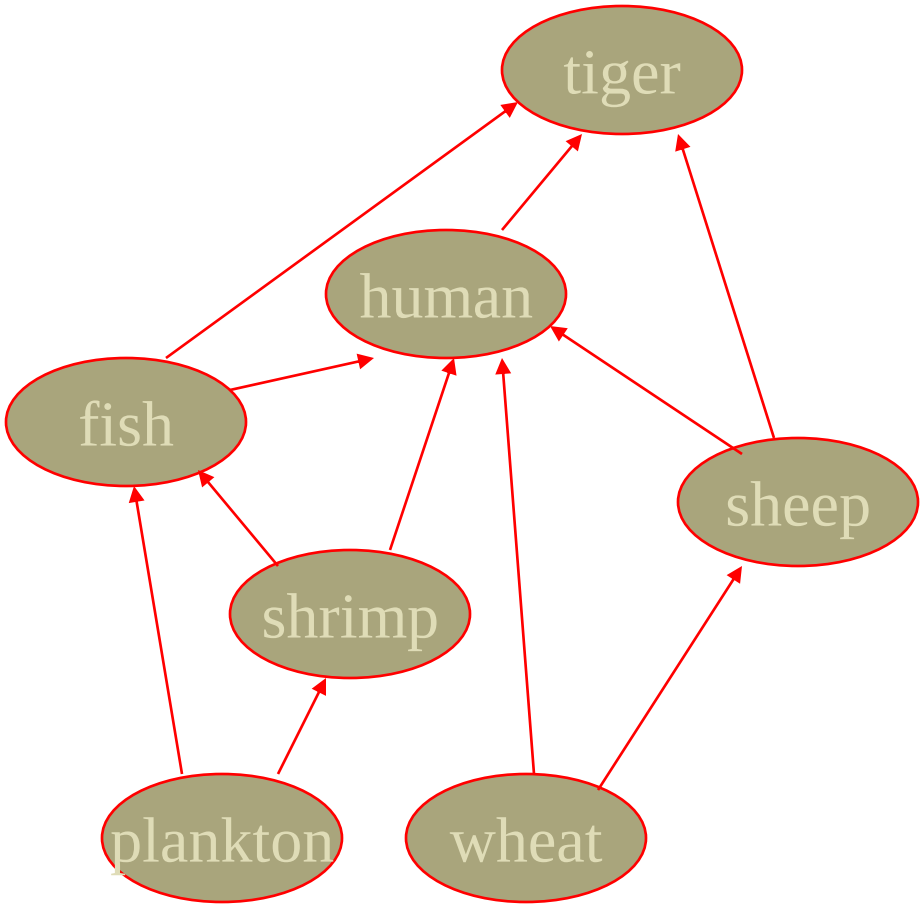
not a dag

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 be possible.

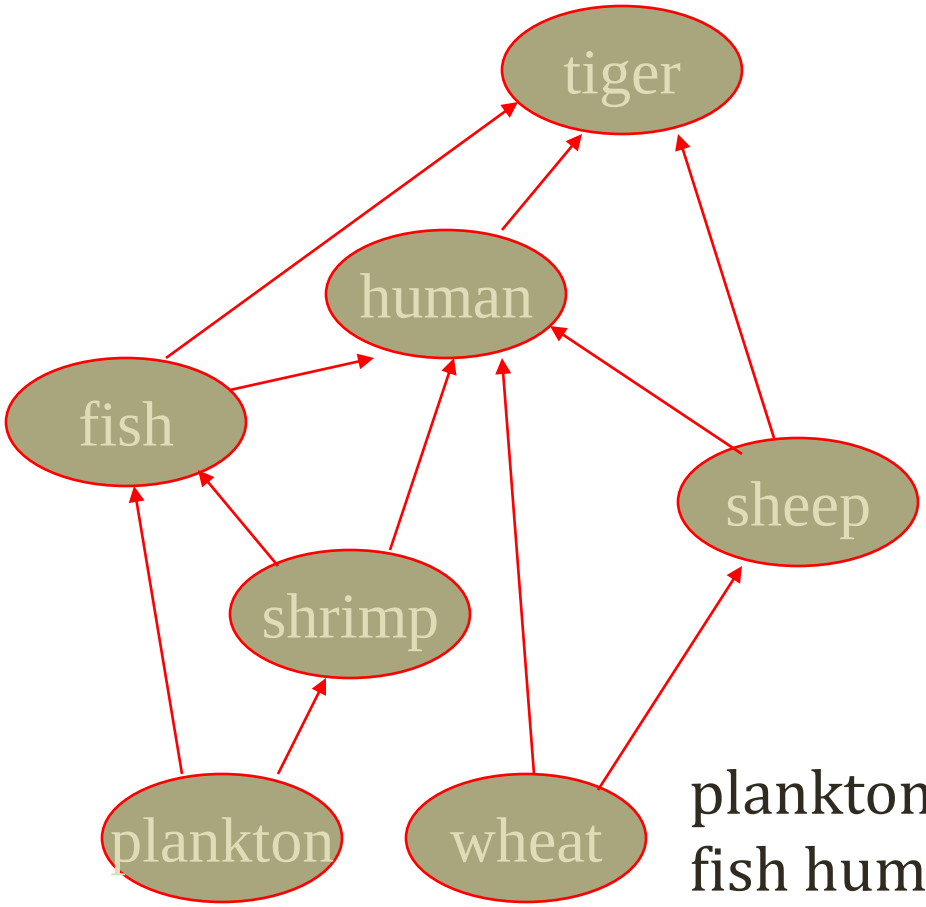
Topological Sorting Example

Order the following items in a food chain



Topological Sorting Example

Order the following items in a food chain



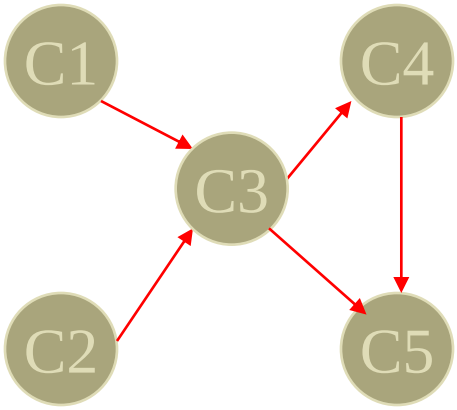
plankton wheat sheep shrimp fish human tiger

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:



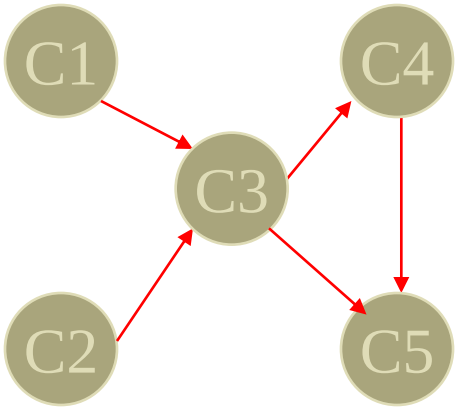
Efficiency:

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:



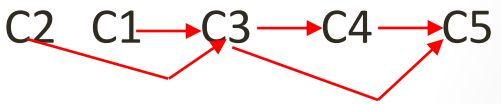
Stack

C5
C4
C3
C1 C2

Popping-off order:

C5, C4, C3, C1, C2

Topologically sorted list:



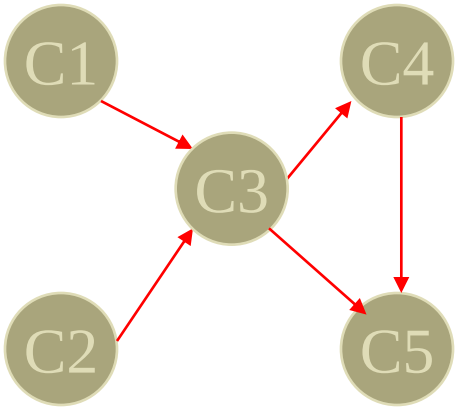
Efficiency: same as DFS traversal

Source Removal Algorithm

Source removal algorithm

- decrease(-by-one)-and-conquer technique
- 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 (problem is solved) or
  - there is no source among remaining vertices (not a dag)

Example:

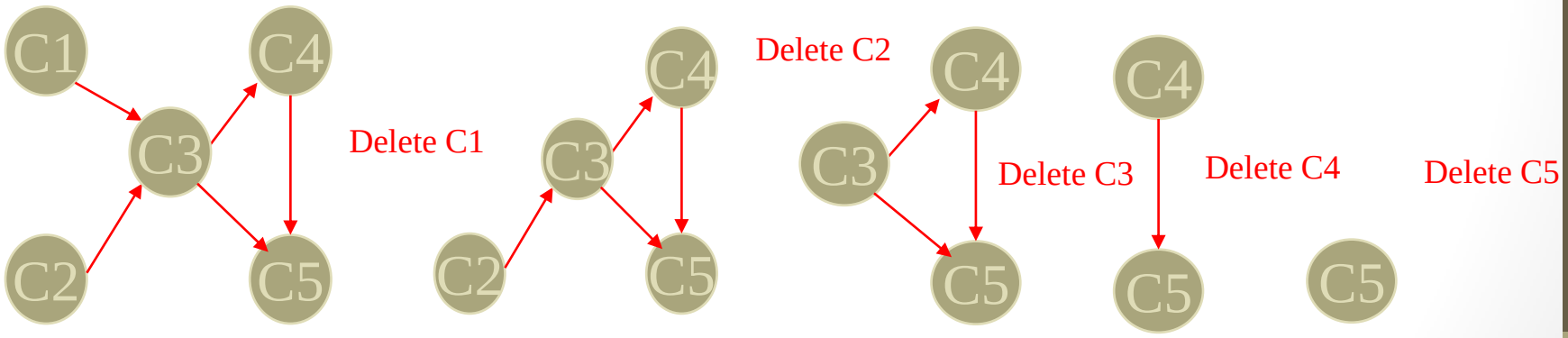


Source Removal Algorithm

Source removal algorithm

- decrease(-by-one)-and-conquer technique
- 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 (problem is solved) or
  - there is no source among remaining vertices (not a dag)

Example:



Efficiency: same as efficiency of the DFS-based algorithm

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 the method of bisection
- Exponentiation by squaring
- Multiplication à la russe (Russian peasant method)
- Fake-coin puzzle
- Josephus problem

Russian Peasant Multiplication

The problem: Compute the product of two positive integers

The idea: Use only operations of halving, doubling and adding

- Can be solved by a decrease-by-half algorithm based on the following formulas.

For even values of n:

$$n * m = \frac{n}{2} * 2m \quad \textit{(recursively)}$$

For odd values of n:

$$n * m = \frac{n-1}{2} * 2m + m \quad \text{if } n > 1$$
$$= m \quad \text{if } n = 1$$

Example of Russian Peasant Multiplication

Compute 20 * 26

n	*m*	
20	26	
10	52	
5	104	
2	208	+ 104
1	416	= **520**

Method reduces to adding *m*'s values corresponding to odd *n*'s.

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.

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.

Decrease by factor 2 algorithm – 2 piles of $\lfloor n/2 \rfloor$ coins each

$O(\log_2 n)$

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.

Decrease by factor 2 algorithm

Decrease by factor 3 algorithm – 3 piles of $\lfloor n/3 \rfloor$ coins each

$O(\log_3 n)$

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
- Partition-based algorithm for selection problem
- Interpolation search
- Some algorithms on binary search trees
- Nim and Nim-like games

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, 12) = gcd(12,0) = 12

One can prove that the size, measured by the second number, decreases at least by half after two consecutive iterations. Hence, $T(n) \in O(\log n)$

Selection Problem

Find the *k*-th smallest element in a list of *n* numbers

- *k* = 1: smaller
- *k* = *n: largest*

- *k* = $\lceil n/2 \rceil$: *median (mediana)*: greater than one half and smaller than the other half

  Example: 4, 1, 10, 9, 7, 12, 8, 2, 15 *median* = ?

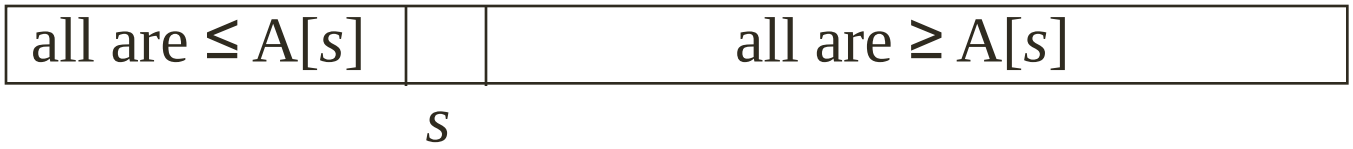
The median is used in statistics as a measure of an average value of a sample. In fact, it is a better (more robust) indicator than the mean (média), which is used for the same purpose.

Algorithms for the Selection Problem

The sorting-based algorithm: Sort and return the *k*-th element
Efficiency (if sorted efficiently): $\Theta(n \log n)$

A faster algorithm is based on using the quicksort-like partition of the list.

Let *s* be a split position obtained by a partition:



Assuming that the list is indexed from 1 to *n*:

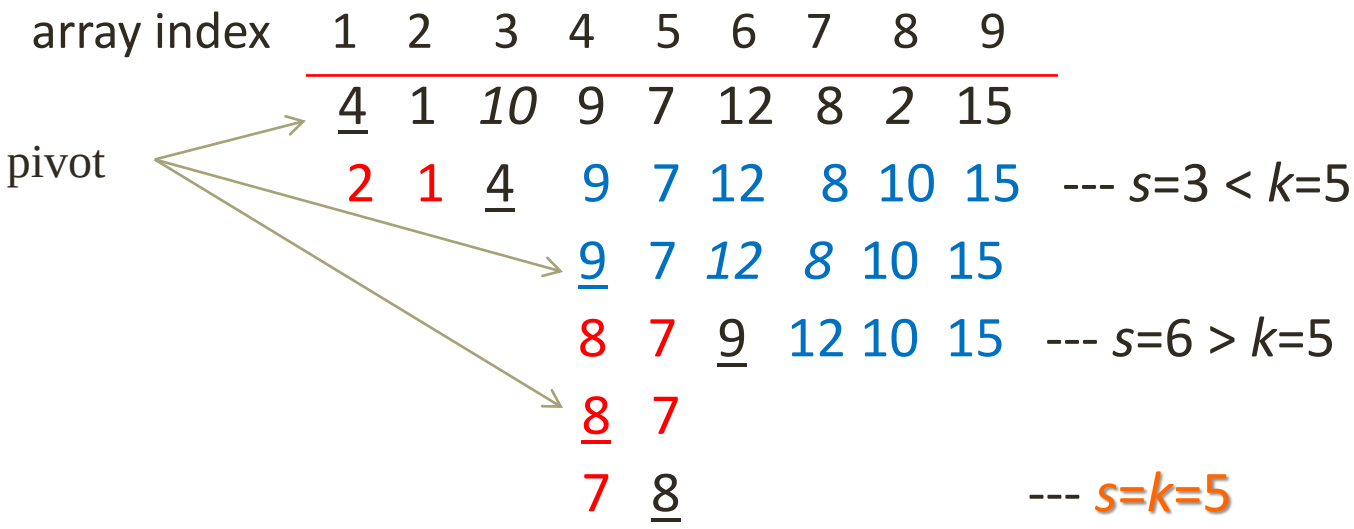
If $s = k$, the problem is solved;

if $s > k$, look for the *k*-th smallest elem. in the left part;
if $s < k$, look for the $(k-s)$-th smallest elem. in the right part.

Note: The algorithm can simply continue until $s = k$.

Tracing the Median / Selection Algorithm

Example: 4 1 10 9 7 12 8 2 15 Here: $n = 9$, $k = \lceil 9/2 \rceil = 5$



Solution: median is 8

Efficiency of the Partition-based Algorithm

Average case (average split in the middle):

$C(n) = C(n/2)+(n+1)$ $\quad\quad$ $C(n) \in \Theta(n)$

(work on just one half + work for placing the pivot)

Worst case (degenerate split): $C(n) \in \Theta(n^2)$

(as quicksort)

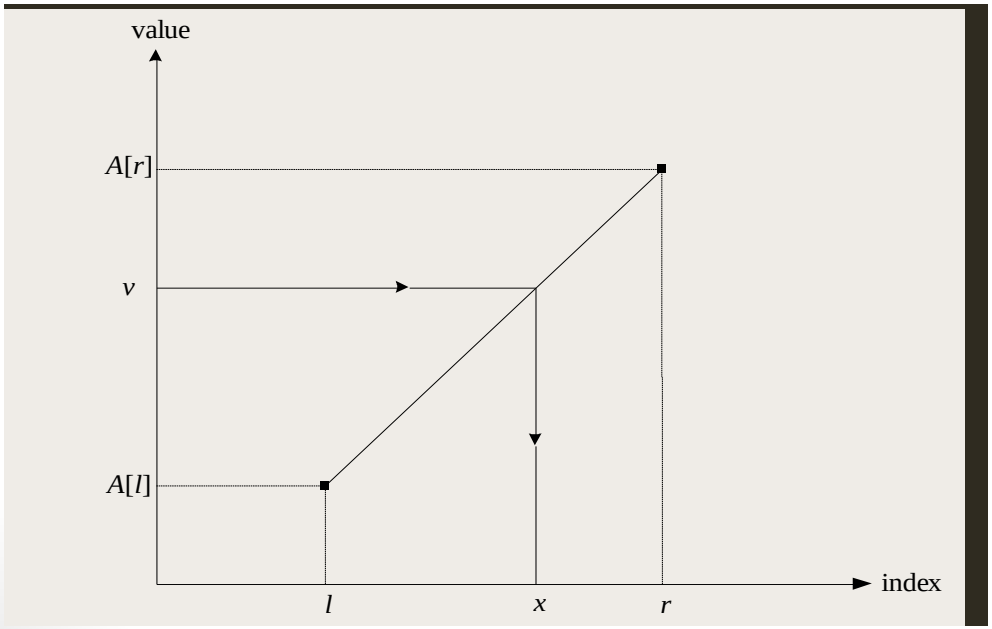
A more sophisticated choice of the pivot leads to a complicated algorithm with $\Theta(n)$ worst-case efficiency.

Interpolation Search

Searches a sorted array similar to binary search but estimates location of the search key in $A[l..r]$ by using its value v.

Mimics human search for "Brown" and "Smith" in a dictionary

Specifically, the values of the array's elements are **assumed** to grow linearly from $A[l]$ to $A[r]$ and the location of v is estimated as the x-coordinate of the point on the straight line through (l, A[l]) and (r, A[r]) whose y-coordinate is v:



Line Equation:

$$\frac{A[r] - A[l]}{r - l} = \frac{v - A[l]}{x - l}$$

$$x = l + \lfloor (v - A[l])(r - l)/(A[r] - A[l]) \rfloor$$

Analysis of Interpolation Search

- Efficiency

  average case: $C(n) < \log_2 \log_2 n + 1$

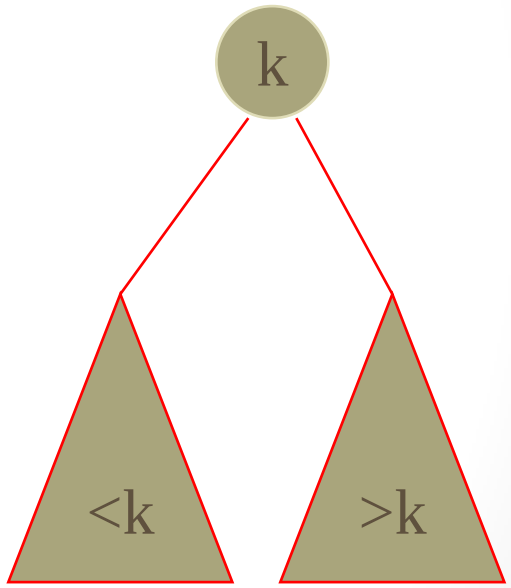
  worst case: $C(n) = n$

- [R. Sedgewick] Preferable to binary search only for VERY large arrays and/or expensive comparisons

Binary Search Tree Algorithms

Several algorithms on BST requires recursive processing of just one of its sub trees, e.g.,

- Searching
- Insertion of a new key
- Finding the smallest (or the largest) key



Searching in Binary Search Tree

Algorithm *BTS*(*x*, *v*)

//Searches for node with key equal to *v* in BST rooted at node *x*

if *x* = NIL return -1

else if *v* = *K*(*x*) return *x*

else if *v* < *K*(*x*) return *BTS*(*left*(*x*), *v*)

else return *BTS*(*right*(*x*), *v*)

Efficiency

worst case: $C(n) = n$

average case (random keys): $C(n) \approx 2\ln n \approx 1.39\log_2 n$