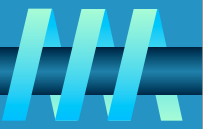
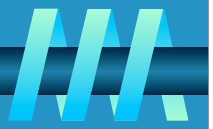
Chapter 4

Decrease and Conquer



- 4.1 Insertion Sort
- 4.2 Topological Sorting
- 4.3 Algorithms for Generating Combinatorial Objects
- 4.4 Decrease-by-a-Constant-Factor Algorithms
- 4.5 Variable-Size-Decrease Algorithms

Decrease-and-Conquer

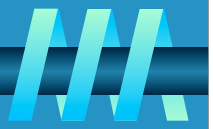


Exploit the relationship between a solution to a given instance of a problem and a solution to its smaller instance.

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**

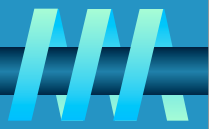
- **Can be implemented either top-down or bottom-up**
- **Also referred to as *inductive* or *incremental* approach**

3 Types of Decrease and Conquer



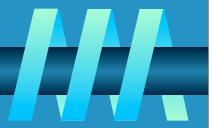
- ***Decrease by a constant*** **(usually by 1):**
  - **insertion sort**
  - **topological sorting**
  - **algorithms for generating permutations, subsets**
- ***Decrease by a constant factor*** **(usually by half)**
  - **binary search and bisection method**
  - **exponentiation by squaring**
  - **multiplication à la russe**
- ***Variable-size decrease*** **(on each iteration...)**
  - **Euclid's algorithm (GCD)**
  - **selection by partition**
  - **Nim-like games**

What's the difference?



Consider the problem of exponentiation: Compute a^n

- **Brute Force:**
  - $a^n = a*a*a*a*...*a$
- **Divide and conquer:**
  - $a^n = a^{n/2} * a^{n/2}$ (more accurately, $a^n = a^{\lfloor n/2 \rfloor} * a^{\lceil n/2 \rceil}$)
- **Decrease by one:**
  - $a^n = a^{n-1} * a$
- **Decrease by constant factor:**
  - $a^n = (a^{n/2})^2$



- **4.1 Insertion Sort**
- **4.2 Topological Sorting**
- **4.3 Algorithms for Generating Combinatorial Objects**
- **4.4 Decrease-by-a-Constant-Factor Algorithms**
- **4.5 Variable-Size-Decrease Algorithms**

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	6 is considered sorted array here (1st iteration)
4 6 | 1 8 5	4 inserted in relation to already sorted array
1 4 6 | 8 5	1 inserted in relation to already sorted array
1 4 6 8 | 5	8 inserted in relation to already sorted array
1 4 5 6 8	5 inserted in relation to already sorted array

| separates "sorted | from unsorted" and underscored is next item to insert

Pseudocode of Insertion Sort

ALGORITHM *InsertionSort*$(A[0..n-1])$

Increasing Order Insertion Sort

//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** ← go through through entire list (array)

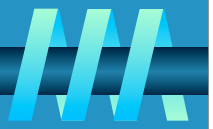
~n times

  $v \leftarrow A[i]$ ← temp storage of value
  $j \leftarrow i-1$ ← set starting point for "unsorted" array of elements
  while $j \geq 0$ **and** $A[j] > v$ **do** ← this is key comparison

~n times

    $A[j+1] \leftarrow A[j]$ ← shuffle values in sorted list, high to low
    $j \leftarrow j-1$ ← reset j to consider the next value in list
  $A[j+1] \leftarrow v$ ← insert into sorted position

Insertion Sort: Time Efficiency



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

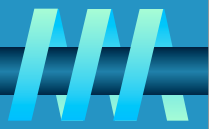
- when input is array of strictly decreasing values (sorted high to low)
- why? because algorithm looks at sorted list from high-to-low and if unsorted is strictly decreasing, then inner loop will have to run entirely through sorted list each time to insert lower number at head of sorted list.

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

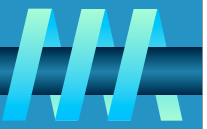
$C_{best}(n) = n - 1 \in \Theta(n)$ (also fast on "almost" sorted arrays)

- when input is array of strictly increasing values (sorted low to high)
- why? because algorithm will insert at the first insertion point in newly sorted list each time. essentially walks "unsorted" list into "sorted" list

Insertion Sort: Analysis



- Space efficiency: in-place
- Stability: yes
  - Stable if preserves the relative order of any two equal elements in its inputs. (1.3, page 19 text) e.x. alpha sorted list of students, sorted on GPA will now have students with same GPA still in alpha order.
    - why? Think about how insertion sort places next element to be sorted after those higher but before those of equal value in the sorted side of the list.
      - how could you change insertion sort to make it so that is not considered stable? (i.e. inserts at another position)
        - » why would you do this?
- Best of elementary sorting algorithms
  - compared to selection sort and bubble sort
- Extension of insertion sort: shellsort
  - better performance on larger data sets

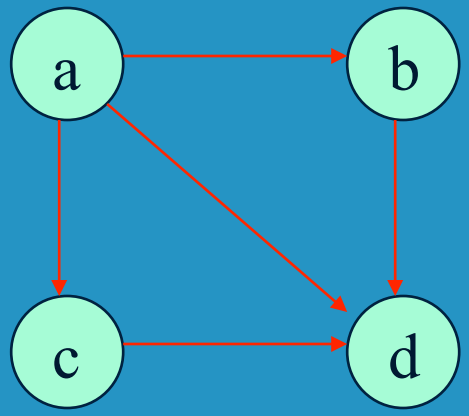


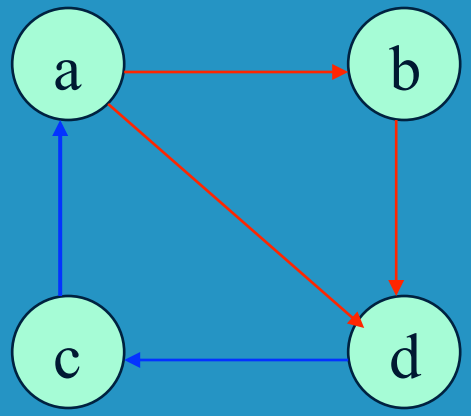
- 4.1 Insertion Sort
- 4.2 Topological Sorting
- 4.3 Algorithms for Generating Combinatorial Objects
- 4.4 Decrease-by-a-Constant-Factor Algorithms
- 4.5 Variable-Size-Decrease Algorithms

Dags and Topological Sorting

A ***dag***: a **d**irected **a**cyclic **g**raph, i.e. a directed graph with no (directed) cycles

a dag





not a dag

WHY?

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.

Directed Graph: Digraph

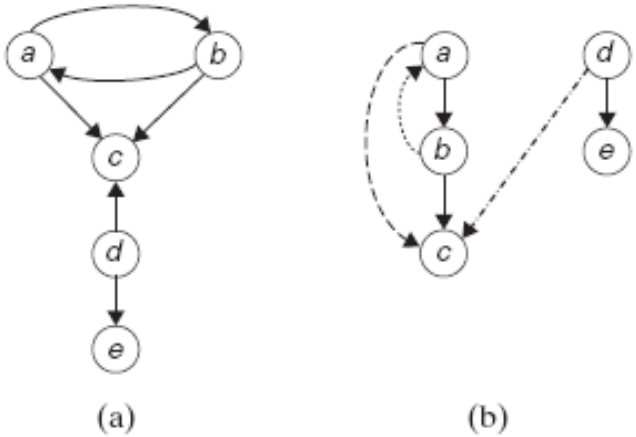


FIGURE 4.5 (a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at a.

Adjacency Matrix

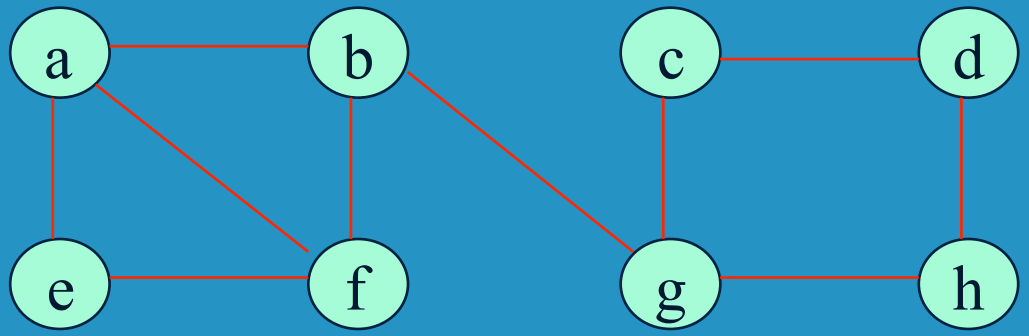
	a	b	c	d	e
a	0	1	1	0	0
b	1	0	1	0	0
c	0	0	0	0	0
d	0	0	1	0	1
e	0	0	0	0	0

Notice that adjacency matrix is not necessarily symmetric. Unlike the undirected graph of chapter 3.

Adjacency List

a	b	c
b	a	c
c		
d	c	e
e		

From Chpt3: DFS traversal of undirected graph



Adjacency Matrix

	a	b	c	d	e	f	g	h
a	0	1	0	0	1	1	0	0
b	1	0	0	0	0	1	1	0
c	0	0	0	1	0	0	1	0
d	0	0	1	0	0	0	0	1
e	1	0	0	0	0	1	0	0
f	1	1	0	0	1	0	0	0
g	0	1	1	0	0	0	0	1
h	0	0	0	1	0	0	1	0

Notice that adjacency matrix is symmetric.

This is because links in undirected graph are considered bidirectional.

Adjacency List

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

Depth First Search (DFS) forest...

Traverse using alphabetical order of vertices (as within reach)

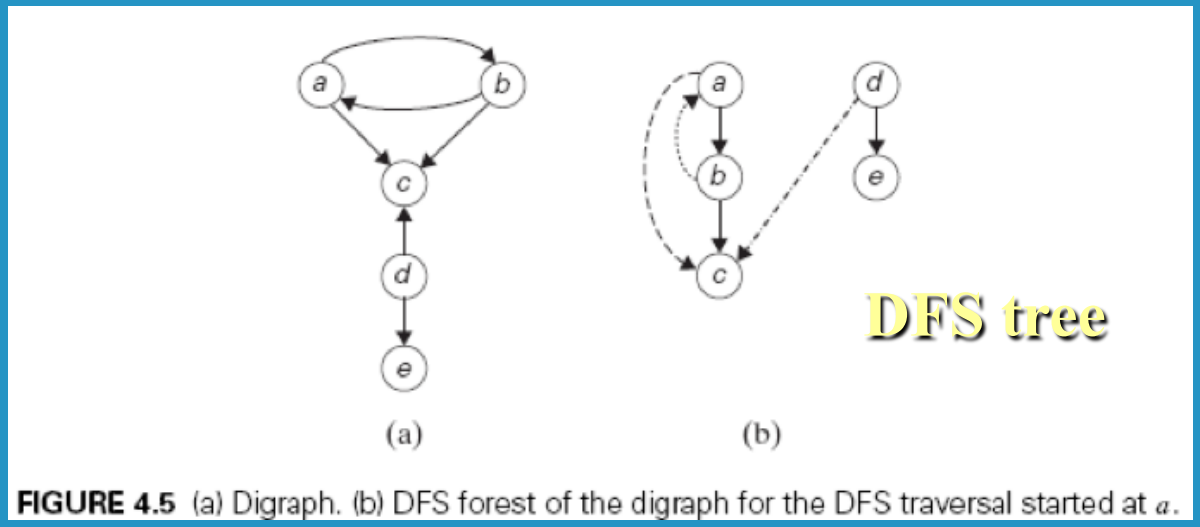


FIGURE 4.5 (a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at *a*.

DFS traversal stack:

$c_{3,1}$
$b_{2,2}$
$a_{1,3}$ ← 1st pushed on stack

$e_{5,4}$
$d_{4,5}$

Reminder...
$X_{push,pop}$

Depth First Search (DFS) forest...

Traverse using alphabetical order of vertices (as within reach)

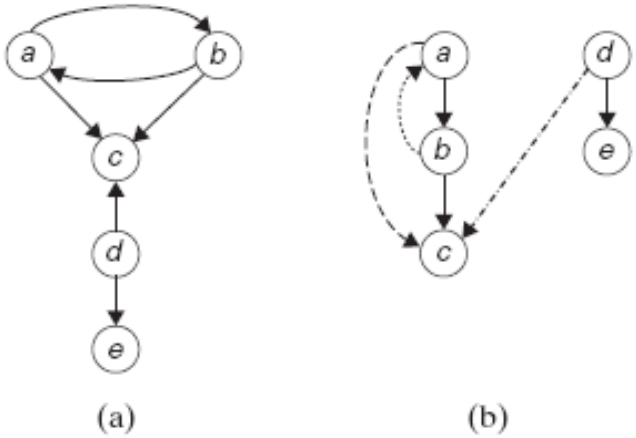


FIGURE 4.5 (a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at a.

Remember using our rules for DFS traversal (lower to higher ranking traversal)

- tree edges: (ab, bc, de)
- back edges: (ba) - those to their ancestor
- forward edges: (ac) – those to their descendants in tree other than their children
  - caused by following our DFS traversal rules (lower to higher ranking traversal)
- cross edges: (dc) – those other than the above...

- directed cycle: (aba) – sequence of 3 or more vertices that starts and ends with the same vertex and in which every vertex is connected to its immediate predecessor by an edge directed from the predecessor to the successor.

- directed acyclic graph DAG/dag: a diagraph with no back edges (no cycles)

Advising time at TTU...

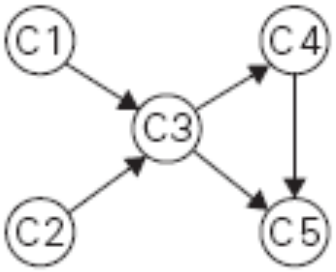


FIGURE 4.6 Digraph representing the prerequisite structure of five courses.

- C1 and C2 have no prerequisites
- C3 requires C1 and C2
- C4 requires C3
- C5 requires C3 and C4

{can only take 1-course per term}

Topological Sorting –or- Topological Ordering

(same definition - said two ways... / TS/TO may be used interchangeably)

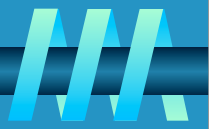
TS/TO of a directed graph (digraph) is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering.

TS/TO of a directed graph (digraph) is a ordering of its vertices such that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends.

For TS/TO to be possible, the diagraph must be a dag.
{reminder: directed acyclic graph has no cycles or loops possible}

- Two efficient algorithms can verify that a digraph is a dag
  - DFS: depth first search
  - source-removal algorithm

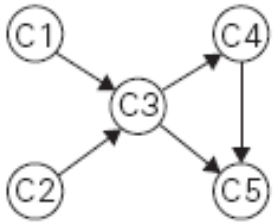
TS/TO Uses



- large projects with numerous perquisites
  - scheduling your courses at TTU...
  - construction project
  - software projects
  - launch of a new product
  - legal proceedings with defined stages
- Solving TS/TO will show no cycles and thus is solvable
- How to find optimal solution (cost, time, etc...)
  - use other technique from operations research areas
    - Critical Path Method (CPM)
    - Program Evaluation and Review Technique (PERT)
  - not sufficient information provided in a simple digraph.
    - to solve for optimality will need transition (link/edge) costs associated with moving from one state (vertex) to another state.

Topological Sorting via DFS

Traverse using numeric order of vertices (as within reach)



(a)

$C5_1$
$C4_2$
$C3_3$
$C1_4$ $C2_5$

(b)

The popping-off order:
C5, C4, C3, C1, C2
The topologically sorted list:
C2 C1→C3→C4→C5

(c)

FIGURE 4.7 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

DFS traversal stack:

$C5_{4,1}$
$C4_{3,2}$
$C3_{2,3}$
$C1_{1,4}$ ← 1st pushed on stack $C2_{5,5}$

My version has more complete info than figure 4.7b. Theirs is an implied push ordering.

Reversing the order of those popped off the stack will give the TS/TO.

Course order (or TS/TO) is: C2, C1, C3, C4, C5

Reminder...
$X_{push,pop}$

Topological Sorting via DFS (continued)

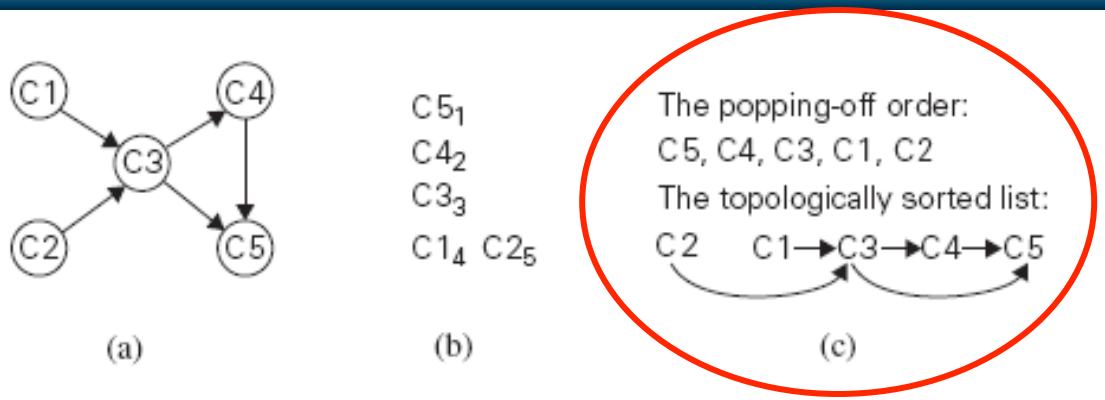
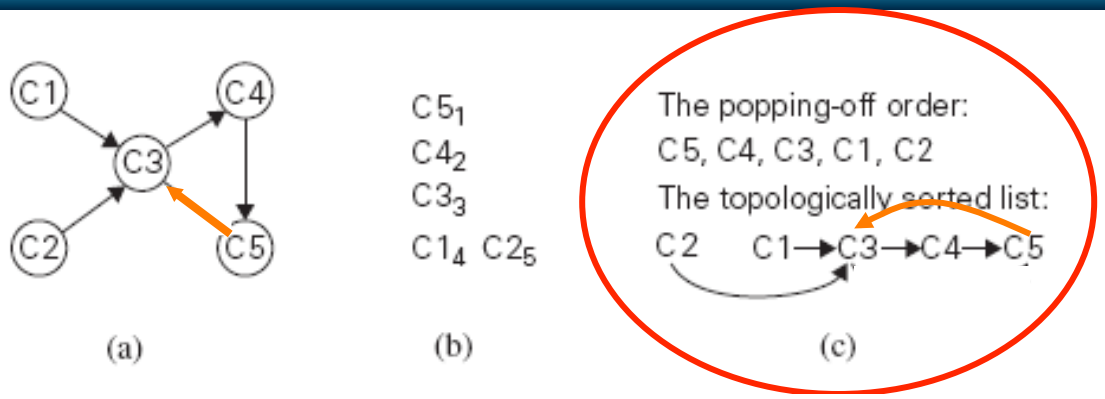


FIGURE 4.7 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

Why does DFS work?
When vertex v is popped off DFS stack, no vertex u with an edge from u to v can be among the vertices popped off before v. (Otherwise (u,v) would have been a back edge, indicating this is a dag and sorting not possible) Therefore, any such vertex u will be listed after v in the popped-off order list, and before v in the reversed list.

Confirmation... (as shown in figure 4.7c)
- **list vertices in order popped off stack**
- **draw edges (links) of diagraph**
- **if all edges point from left-to-right, solution is correct**
- **any edges point from right-to-left, problem... as is not a dag**

Topological Sorting via DFS (continued)



Counter example with NOT a dag

Confirmation... (counter example with dag)

- list vertices in order popped of stack
- draw edges (links) of diagraph
- if all edges point from left-to-right, solution is correct
- any edges point from right-to-left, problem... as is not a dag

Logical explanation here...

- can not have...
  - C3 as a prerequisite of C4
  - C4 is a prerequisite of C5
  - C5 is a prerequisite of C3

Topological Sorting via Source-Removal

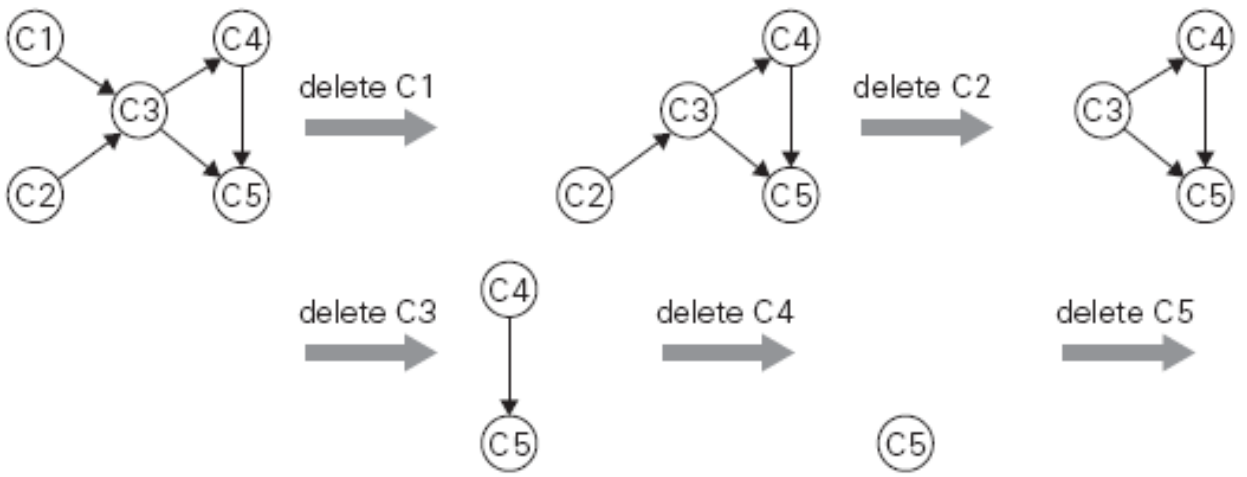


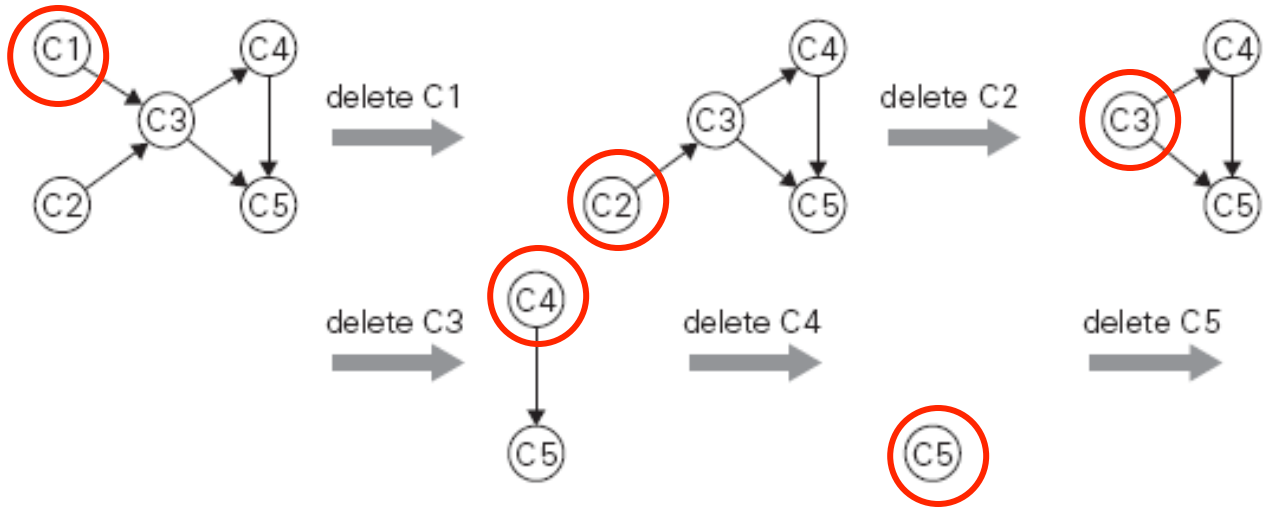
FIGURE 4.8 Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.

Source removal algorithm (decrease-by-one-and-conquer)

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)

Efficiency: same as efficiency of the DFS-based algorithm

Topological Sorting via Source-Removal



The solution obtained is C1, C2, C3, C4, C5

solution could also be: C2, C1, C3, C4, C5

FIGURE 4.8 Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.

decrease-(by one)-and-conquer:

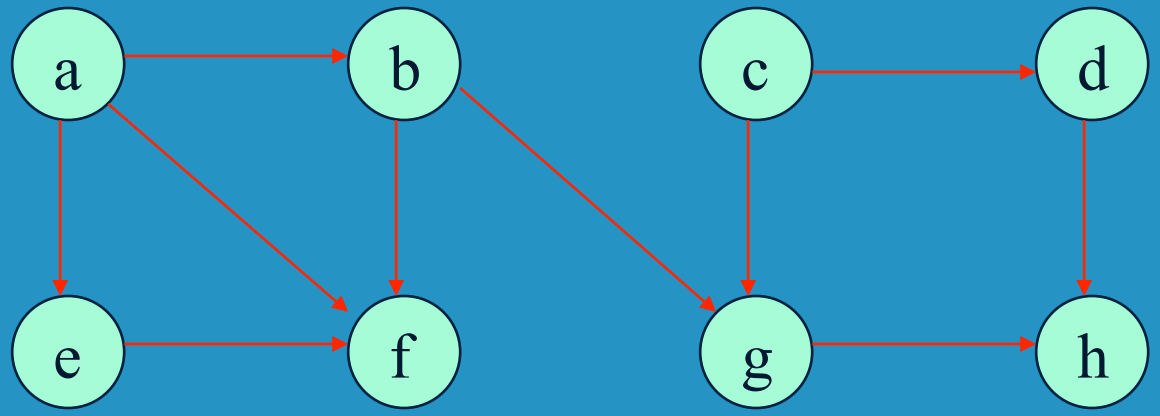
- **identify in diagraph a source (vertex with no incoming edges)**
- **delete that source and all its outgoing edges**

- **if several sources, arbitrarily select one**
- **if no sources, stop as problem can not be solved**

Or start in order specified by some external rule.

ie. lowest number or alphabetic order

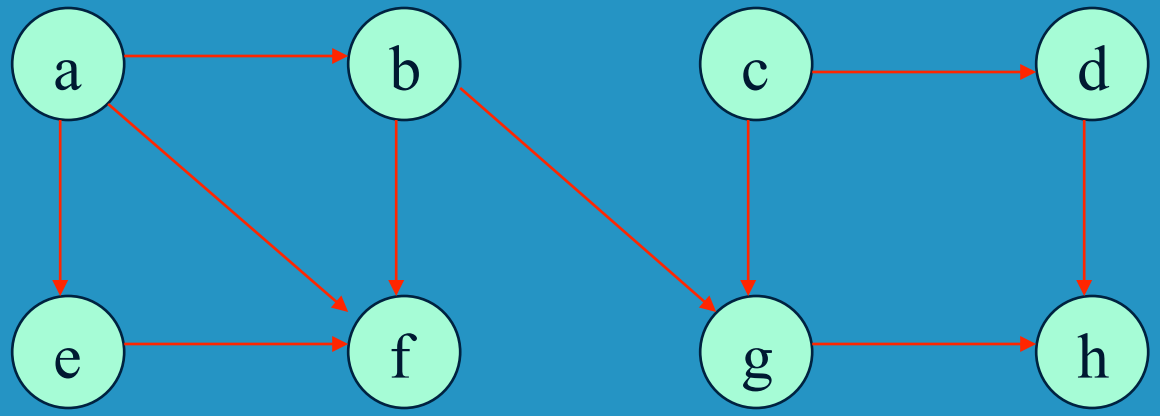
DFS-based Algorithm (another example)



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!

DFS-based Algorithm (another example)



Popping off order: f, h, g, b, e, a, d, c

DFS traversal stack:

$h_{5,2}$

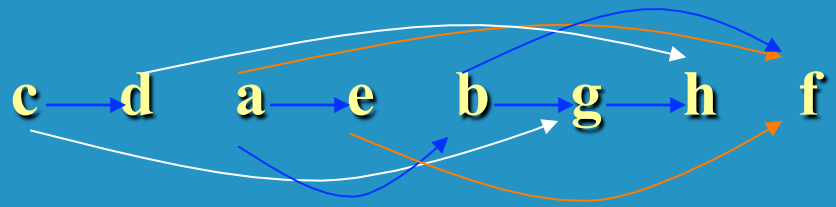
$f_{3,1}$ $g_{4,3}$

$b_{2,4}$ $e_{6,5}$ $d_{8,7}$

1st pushed on the stack $a_{1,6}$ $c_{7,8}$

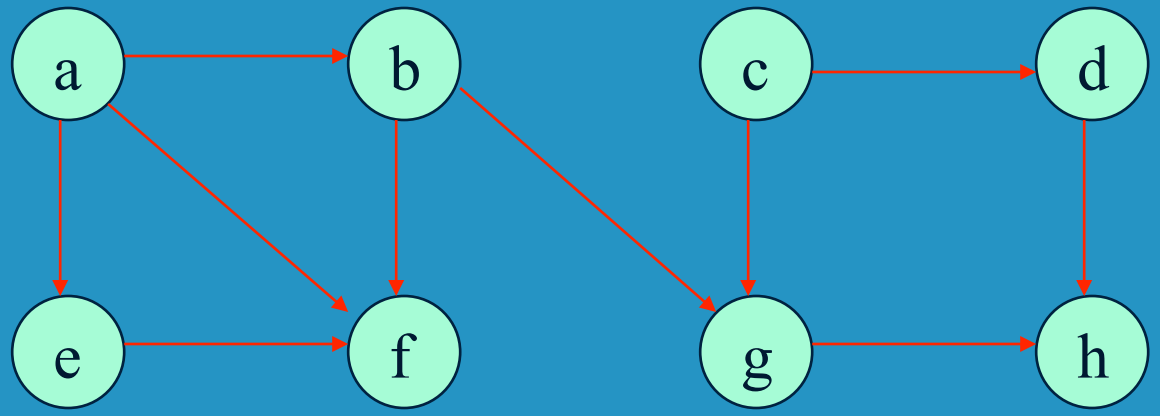
vertex $_{\text{push, pop}}$

TS/TO is the...
Topologically Sorted List:
(reverse of popoff order)



colors of lines just for clarity of crossing links here

Source Removal Algorithm (another example)

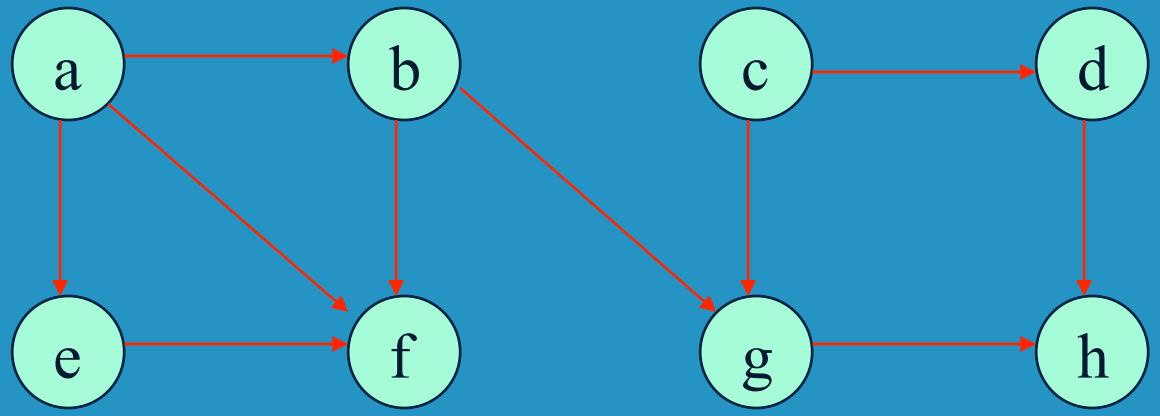


Source removal algorithm

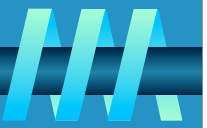
Repeatedly identify and remove a *source* (a vertex with no incoming edges) and all the edges incident to it until either no vertex is left (problem is solved) or there is no source among remaining vertices (not a dag)

Efficiency: same as efficiency of the DFS-based algorithm

Source Removal Algorithm (another example)



solution 1	solution 2	solution 2	solution n
- delete a - delete b - delete e - delete f - delete c - delete d - delete g - delete h	- delete a - delete e - delete b - delete f - delete c - delete g - delete d - delete h	- delete c - delete d - delete a - delete b - delete g - delete h - delete e - delete f	every solution will start with either node a or node c why? no incoming edges to only those two
solution: a b e f c g h	solution: a e b f c g d h	solution: c d a b g h e f	solution: others...



- 4.1 Insertion Sort
- 4.2 Topological Sorting
- 4.3 Algorithms for Generating Combinatorial Objects
- 4.4 Decrease-by-a-Constant-Factor Algorithms
- 4.5 Variable-Size-Decrease Algorithms

Permutation and Combination Review (1)

Permutation: when the order **does** matter

Combination: when the order **does not** matter

Permutation – 2 types:

- repetition allowed: 2,2,2 is valid
- NO repetition: i.e. race, you can't be both first and fifth

Combination– 2 types:

- repetition allowed: 1,1,5,5,10,10,10 as in coins
- NO repetition: i.e. lottery numbers 9,14, 22, 34,56,unique

In "common" language, combination is typically used improperly. (don't often hear use of permutation on the streets...)

- combination lock: order of digits does matter...
  - order does matter – it is a permutation lock
    - if "combination" to lock is 456, lock will not accept 654 and open...

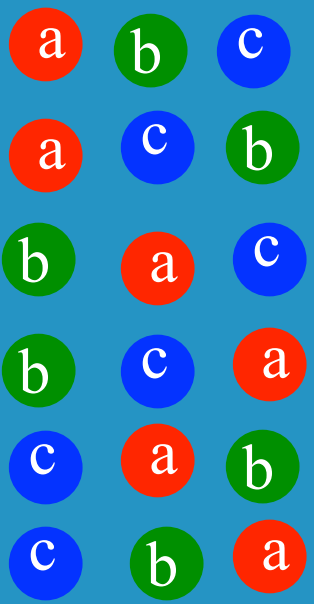
Permutation and Combination Review (2)

Permutations: ways of ordering objects

$${}_3P_3 = 6$$

How many total objects

How many are placed in a row



$${}_nP_r = \frac{n!}{(n-r)!}$$

Order matters

Combinations: ways of choosing objects

$${}_3C_3 = 1$$

How many total objects

How many we are choosing

$${}_nC_r = \frac{n!}{(n-r)!\,r!}$$

Objects we choose matters
Order does NOT matter

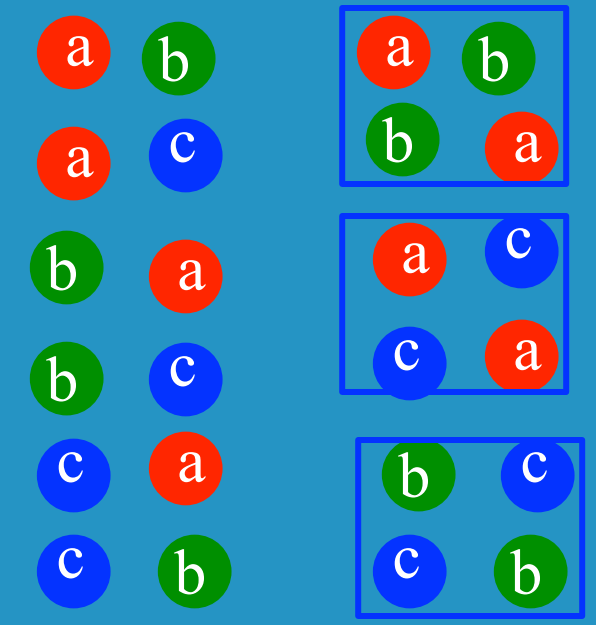
Permutation and Combination Review (3)

Permutations: ways of ordering objects

$${}_3P_2 = 6$$

How many total objects

How many are placed in a row



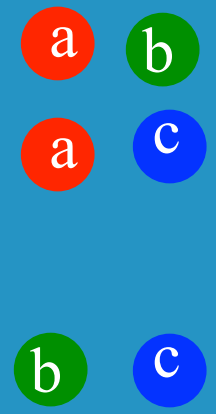
Order matters

Combinations: ways of choosing objects

$${}_3C_2 = 3$$

How many total objects

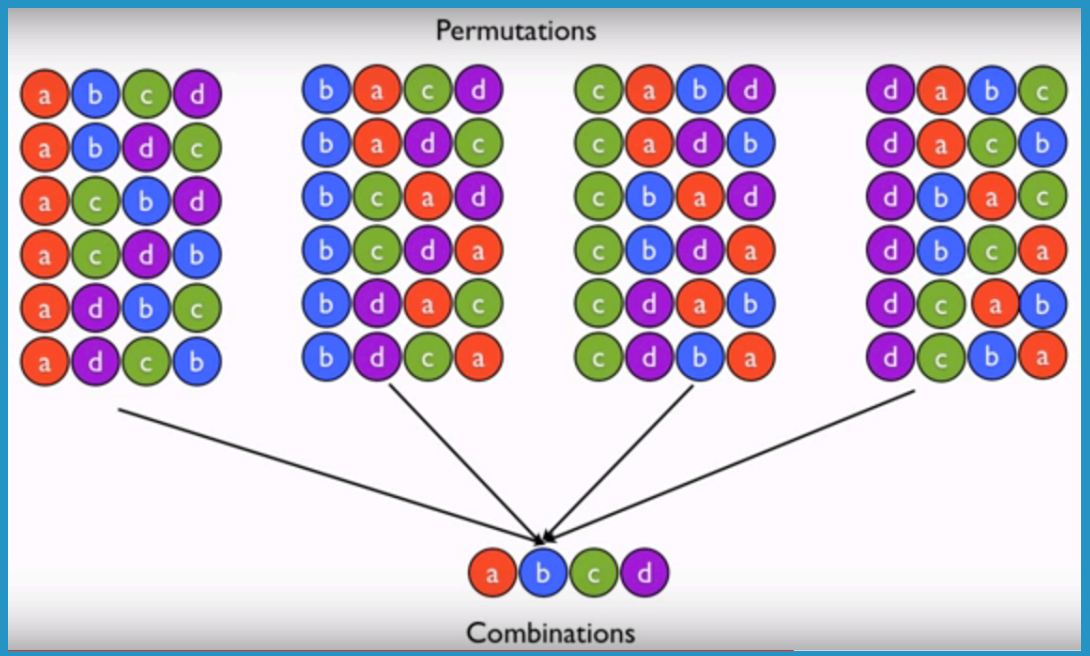
How many we are choosing



Objects we choose matters
Order does NOT matter

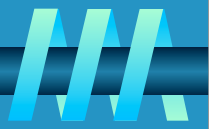
Permutation and Combination Review (4)

$$ {}_nP_r = \frac{n!}{(n-r)!} \qquad {}_4P_4 = \frac{4!}{(4-4)!} = 24 $$



$$ {}_nC_r = \frac{n!}{(n-r)!\,r!} \qquad {}_4C_4 = \frac{4!}{(4-4)!\,4!} = 1 $$

Generating Permutations



Minimal-change **decrease-by-one algorithm**

If $n = 1$ return 1; otherwise, generate recursively the list of all permutations of 12…n-1 and then insert n into each of those permutations by starting with inserting n into 12...n-1 by moving right to left and then switching direction for each new permutation

---

Example: n=4

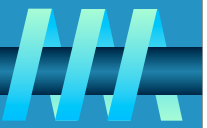
start		1			$_1P_1 = 1$
insert 2 into 1 right to left	12	21			$_2P_2 = 2$
insert 3 into 12 right to left	123	132	312		$_3P_3 = 6$
insert 3 into 21 left to right	321	231	213		
insert 4 into 123 right to left	1234	1243	1423	4123	
insert 4 into 132 left to right	4132	1432	1342	1324	
insert 4 into 312 right to left	3124	3142	3412	4312	
insert 4 into 321 left to right	4321	3421	3241	3214	$_4P_4 = 24$
insert 4 into 231 right to left	2314	2341	2431	4231	
insert 4 into 213 left to right	4213	2413	2143	2134	

Other permutation generating algorithms

- **Johnson-Trotter (p. 145)**
- **Lexicographic-order algorithm (p. 146)**
- **Heap's algorithm (Problem 4 in Exercises 4.3)**

Other techniques...
As an exercise left to the reader...

Potentially good homework or exam questions.



- 4.1 Insertion Sort
- 4.2 Topological Sorting
- 4.3 Algorithms for Generating Combinatorial Objects
- 4.4 Decrease-by-a-Constant-Factor Algorithms
- 4.5 Variable-Size-Decrease Algorithms

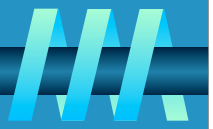
Decrease-by-Constant-Factor Algorithms

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**

Binary Search



Very efficient algorithm for searching in sorted array:

K

vs

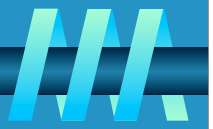
A[0] . . . A[*m*] . . . A[*n*-1]

A[ ] in ascending order

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 ← 0;   r ← n-1
while l ≤ r do                          ← do while indices don't cross
    m ← ⌊(l+r)/2⌋                       ← set middle element
    if  K = A[m]  return m              ← return index of match
    else if K < A[m]   r ← m-1          ← reset the right-side index
         else l ← m+1                   ← reset the left-side index
return -1
```

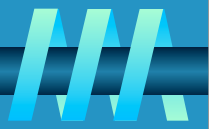
Binary Search: k = 70



index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	l						m						r

$70 > 55$ $k > A[m]$

Binary Search: k = 70



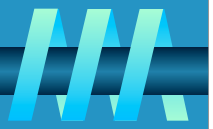
index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	l						m						r

l = m + 1

index								7	8	9	10	11	12
value								70	74	81	85	93	98
iteration 2								l		m			r

k < A[m] 70 < 81

Binary Search: k = 70



index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	l						m						r

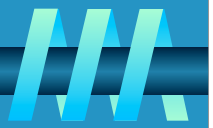
index								7	8	9	10	11	12
value								70	74	81	85	93	98
iteration 2								l		m			r

r = m - 1

index								7	8				
value								70	74				
iteration 3								l,m	r				

k = A[m] 70 = 70

Analysis of Binary Search



- Time efficiency
  - worst-case recurrence: $C_w(n) = 1 + C_w(\lfloor n/2 \rfloor)$ for n>1,
    $C_w(1) = 1$

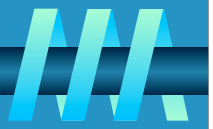
    solution: $C_w(n) = \lceil \log_2(n+1) \rceil$

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

- Optimal for searching a sorted array

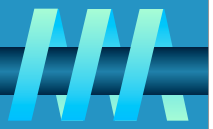
- Limitations: must be a sorted array (not linked list)
  - why?
  - because the traversal process of linked list will eliminate the advantage of halving the search space each time.

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.

Analysis of Fake Coin Puzzle

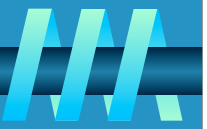


- Decrease by factor 2 algorithm
- Time efficiency
  - worst-case recurrence: $W_w(n) = 1 + W_w(\lfloor n/2 \rfloor)$ for n>1,
    $W_w(1) = 0$

    solution: $W_w(n) = \lfloor \log_2(n+1) \rfloor$
- This is not the most efficient solution to this weighty question...
  - Consider decrease by factor 3 algorithm...
    - See text section 4.4 question 10

- 4.1 Insertion Sort
- 4.2 Topological Sorting
- 4.3 Algorithms for Generating Combinatorial Objects
- 4.4 Decrease-by-a-Constant-Factor Algorithms
- 4.5 Variable-Size-Decrease Algorithms

Variable-Size-Decrease Algorithms

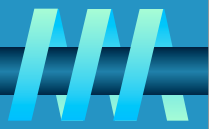


In the variable-size-decrease variation of decrease-and-conquer, instance size reduction varies from one iteration to another

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**

Selection Problem



Problem of finding the *k*-th smallest element in a list of *n* numbers. (called the k-th order statistic)

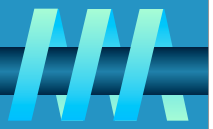
- $k = 1$ or $k = n$

- *median*: $k = \lceil n/2 \rceil$

  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, which is used for the same purpose.

Selection Problem



Problem of finding the *k*-th smallest element in a list of *n* numbers. (called the k-th order statistic)

- $k = 1$ or $k = n$

- *median*: $k = \lceil n/2 \rceil$

  Example: 4, 1, 10, 9, 7, 12, 8, 2, 15 median = $\lceil 9/2 \rceil = 5^{th}$

  1, 2, 4, 7, 8, 9, 10, 12, 15 easier to see if already sorted

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, 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 by mergesort): $\Theta(n \log n)$

A faster algorithm is based on the array *partitioning*:

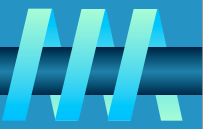
Assuming that the array is indexed from 0 to n-1 and s is a split position obtained by the array partitioning:

If $s = k$-1, the problem is solved;

if $s > k$-1, look for the k^{th} smallest element in the left part;
if $s < k$-1, look for the $(k\text{-}s)^{th}$ smallest element in the right part.

Note: The algorithm can simply continue until $s = k$-1.

Two Partitioning Algorithms

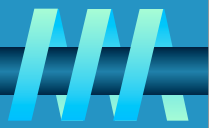


There are two principal ways to partition an array:

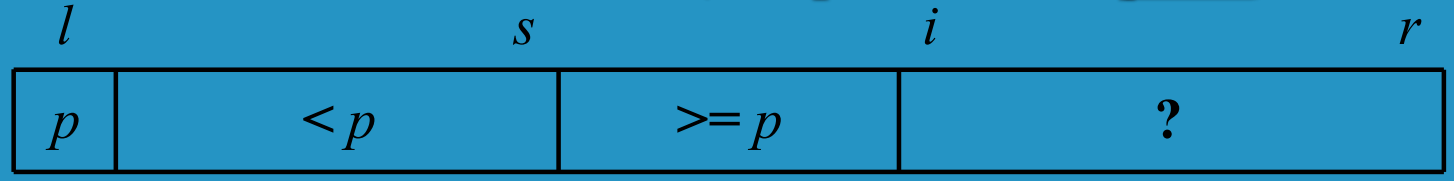
- One-directional scan (Lomuto's partitioning algorithm)
- Two-directional scan (Hoare's partitioning algorithm)

Note: These PARTITION, not sort...

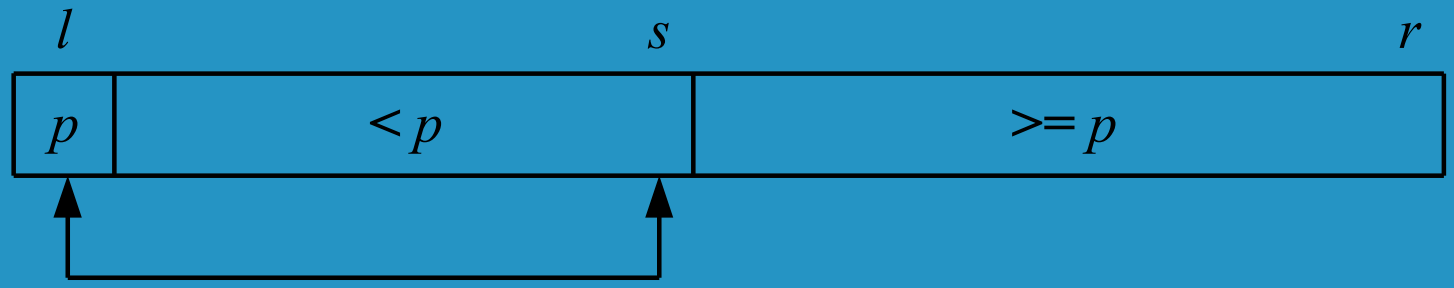
Lomuto's Partitioning Algorithm



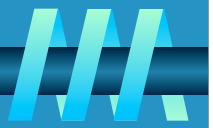
Scans the array left to right maintaining the array's partition into three contiguous sections: $< p$, $\geq p$, and unknown, where p is the value of the first element (the partition's *pivot*).



On each iteration the unknown section is decreased by one element until it's empty and a partition is achieved by exchanging the pivot with the element in the split position s.



Tracing Lomuto's Partioning Algorithm



0	1	2	3	4	5	6	7	8
s	i							
4	1	10	8	7	12	9	2	15
p	s	i						
4	1	10	8	7	12	9	2	15
p	s						i	
4	1	10	8	7	12	9	2	15
p		s						i
4	1	2	8	7	12	9	10	15
p		s						
4	1	2	8	7	12	9	10	15
2	1	4	8	7	12	9	10	15

find the 5th smallest element in the array (k=5)

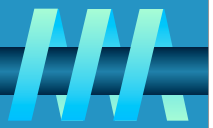
proceeds down array until finds case of A[i]<p

swapped values in A[i] and A[s+1]
increment i and compare again

reposition the pivot point

since s=2 is smaller than k-1=4
do again on right portion of array...

Tracing Lomuto's Partioning Algorithm



0	1	2	3	4	5	6	7	8
			s	i				
			8	7	12	9	10	15
				s	i			
			8	7	12	9	10	15
				s				i
			8	7	12	9	10	15
			7	8	12	9	10	15

find the 5th smallest element in the array (k=5)

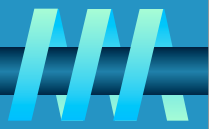
proceeds down array until finds case of A[i]<p but does not find in this example...

reposition the pivot point

median = ⌈9/2⌉ = 5th element in list

since s=k-1=4 we stop, median is value 8

Tracing Quickselect (Partition-based Algorithm)



Find the median of 4, 1, 10, 9, 7, 12, 8, 2, 15

Here: $n = 9$, $k = \lceil 9/2 \rceil = 5$, $k - 1 = 4$ (this is index position)

median = $\lceil 9/2 \rceil = 5^{th}$ element in list

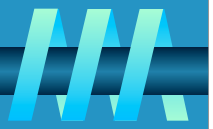
after 1st partitioning: $s = 2 < k - 1 = 4$

after 2nd partitioning: $s = 4 = k - 1$

0	1	2	3	4	5	6	7	8
4	1	10	8	7	12	9	2	15
2	1	**4**	8	7	12	9	10	15
			8	7	12	9	10	15
			7	**8**	12	9	10	15

The median is A[4]= 8

Efficiency of Quickselect



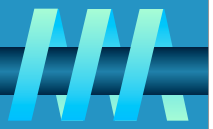
Average case (average split in the middle):

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

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

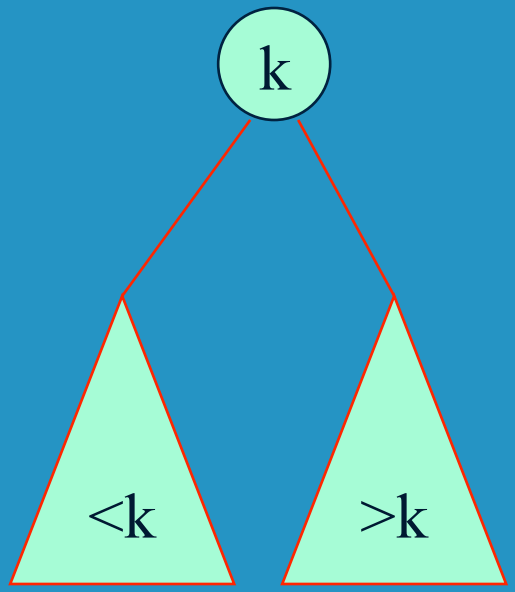
A more sophisticated choice of the pivot leads to a complicated algorithm with $\Theta(n)$ worst-case efficiency.

Binary Search Tree Algorithms

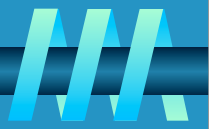


Several algorithms on BST requires recursive processing of just one of its subtrees, e.g.,

- **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: $C(n) \approx 2\ln n \approx 1.39\log_2 n$