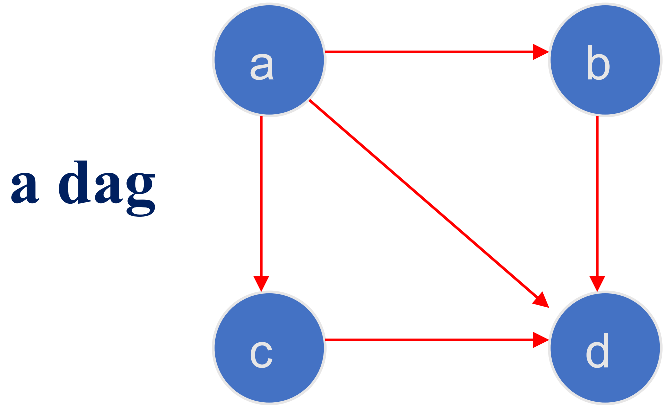
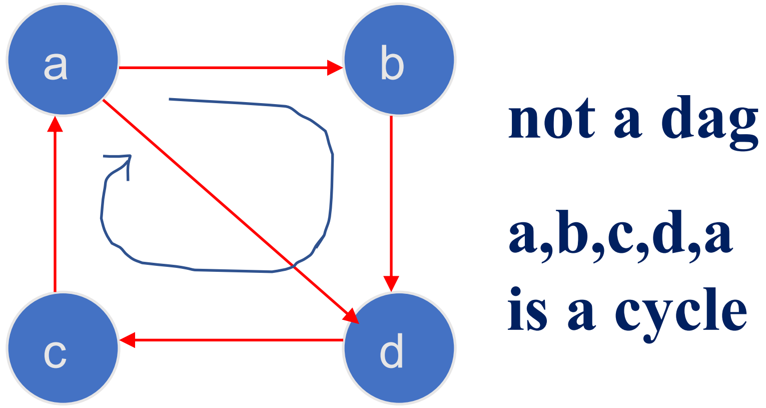
4.2 Topological Sorting

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



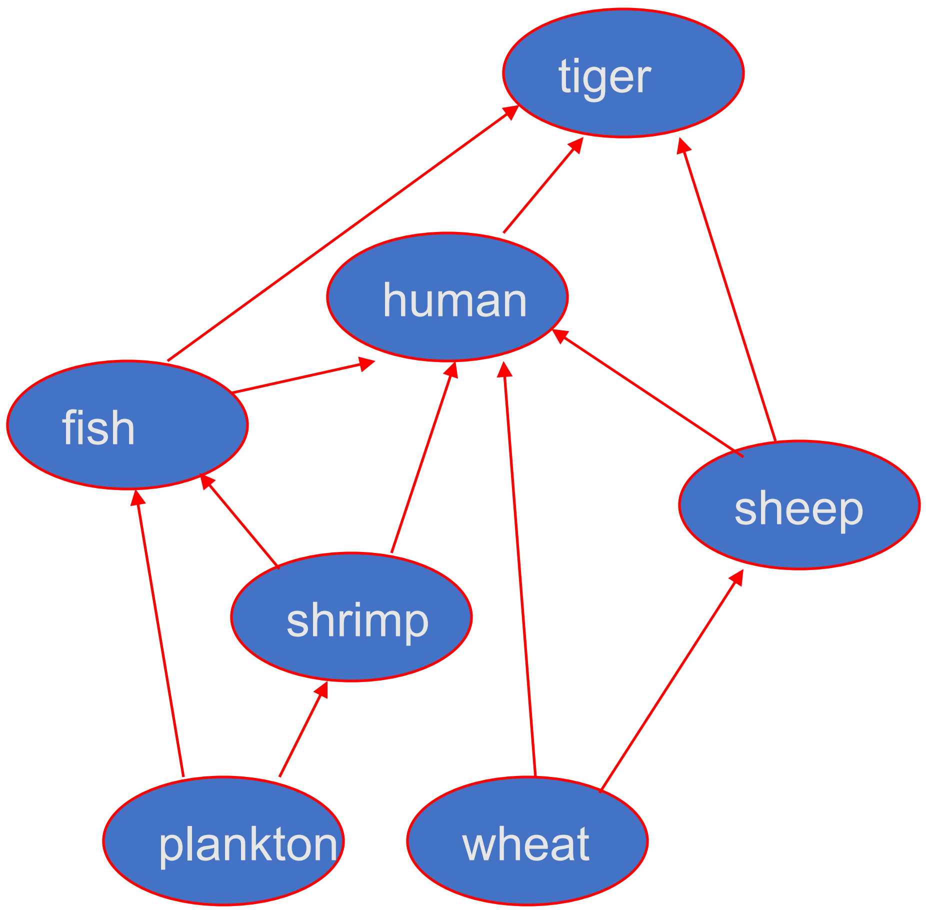


Arise in modeling many problems that involve prerequisite constraints
(construction projects, document version control)

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.

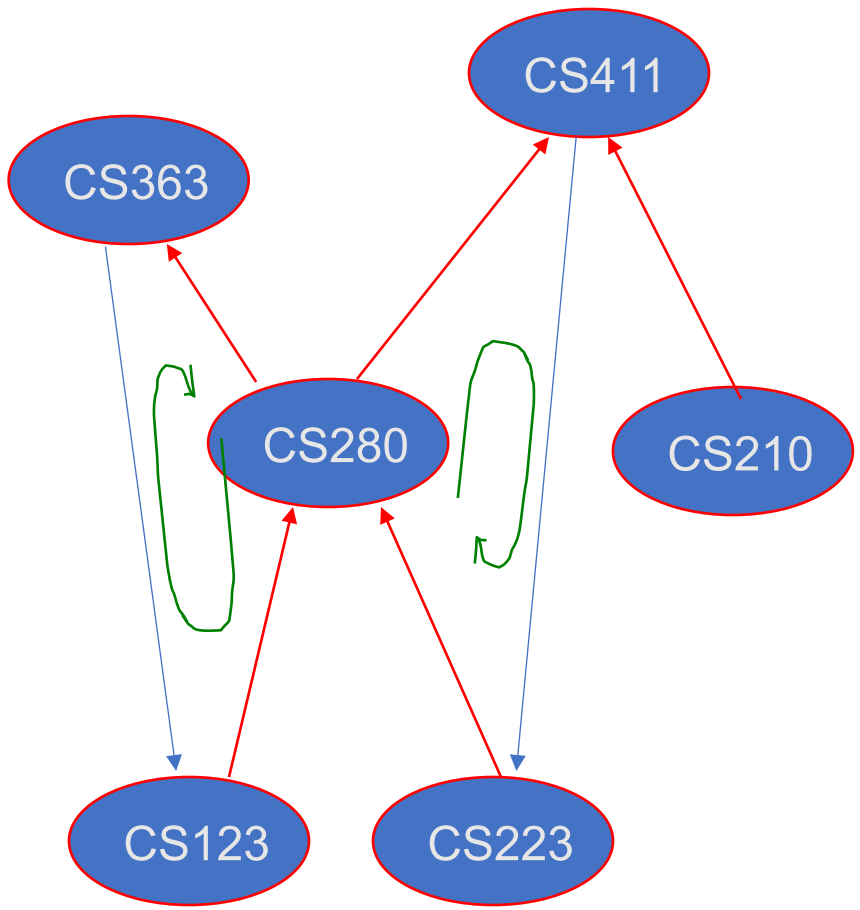
Topological Sorting Example

Order the following items in a food chain



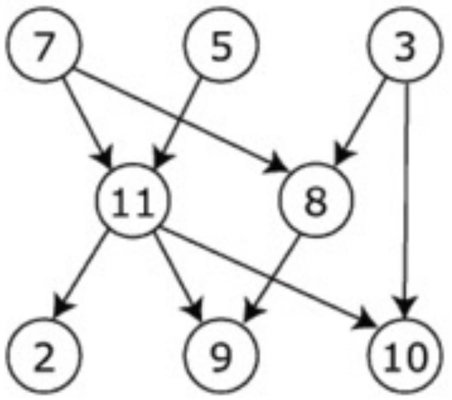
Topological Sorting Example

Order the following courses in a curriculum



Topological Sorting Example

- Topological sorting algorithms were first studied in the early 1960s in the context of the PERT (program evaluation review technique) for scheduling in project management (Jarnagin 1960).
- The jobs are represented by vertices, and there is an edge from *x* to *y* if job *x* must be completed before job *y* can be started.



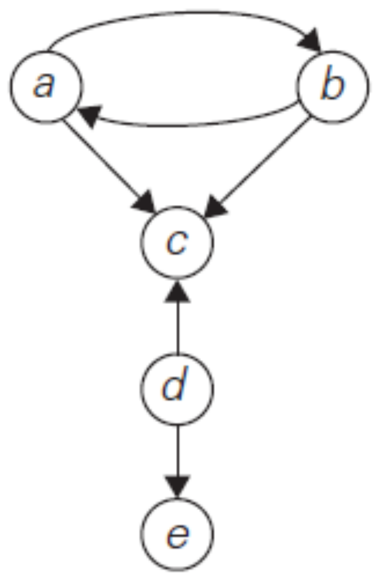
The graph shown to the left has many valid topological sorts, including:

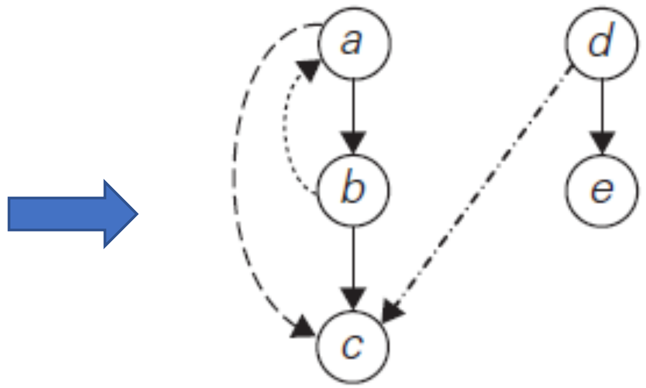
- 7, 5, 3, 11, 8, 2, 9, 10 (visual left-to-right, top-to-bottom)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
- 3, 7, 8, 5, 11, 10, 2, 9
- 5, 7, 3, 8, 11, 10, 9, 2 (fewest edges first)
- 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
- 7, 5, 11, 2, 3, 8, 9, 10

Topological Sorting

- Two algorithms to solve the topological sorting problem (i.e., to determine a directed graph is a dag or not)
  - DFS-based Algorithm
  - Source Removal Algorithm

Depth First Search on a Digraph





What's the order vertices become dead-ends? (i.e., The order vertices are popped off the traversal stack):

c, b, a, e, d

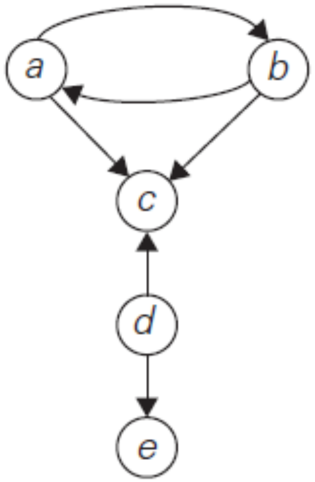
How's the above order related to topological sort?

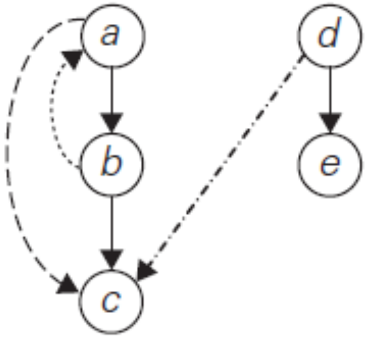
DFS-based Algorithm

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?→ if yes → NOT a dag!

Example:





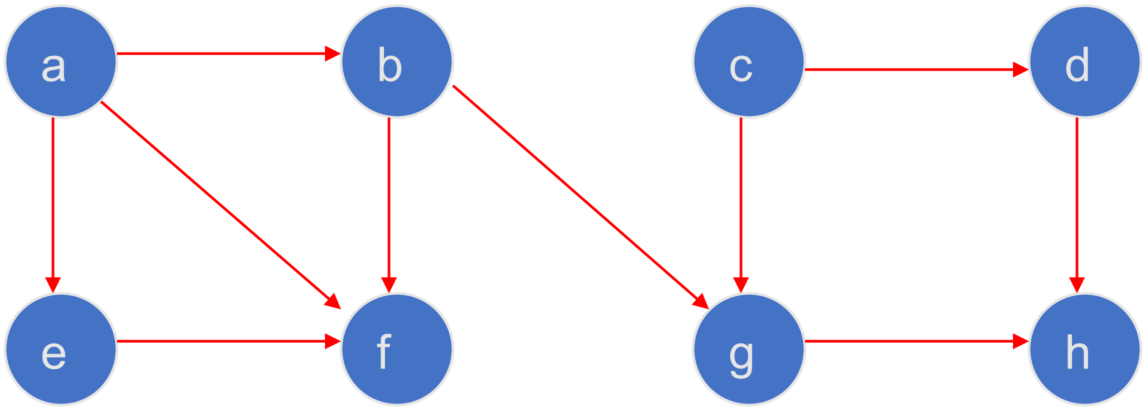
- The order vertices become dead-ends: c, b, a, e, d
- Reverse order: d, e, a, b, c
- Is this a solution? Not yet.
- Back edges encountered? yes → NOT a dag
- ***Not a solution***

DFS-based Algorithm

DFS-based algorithm for topological sorting

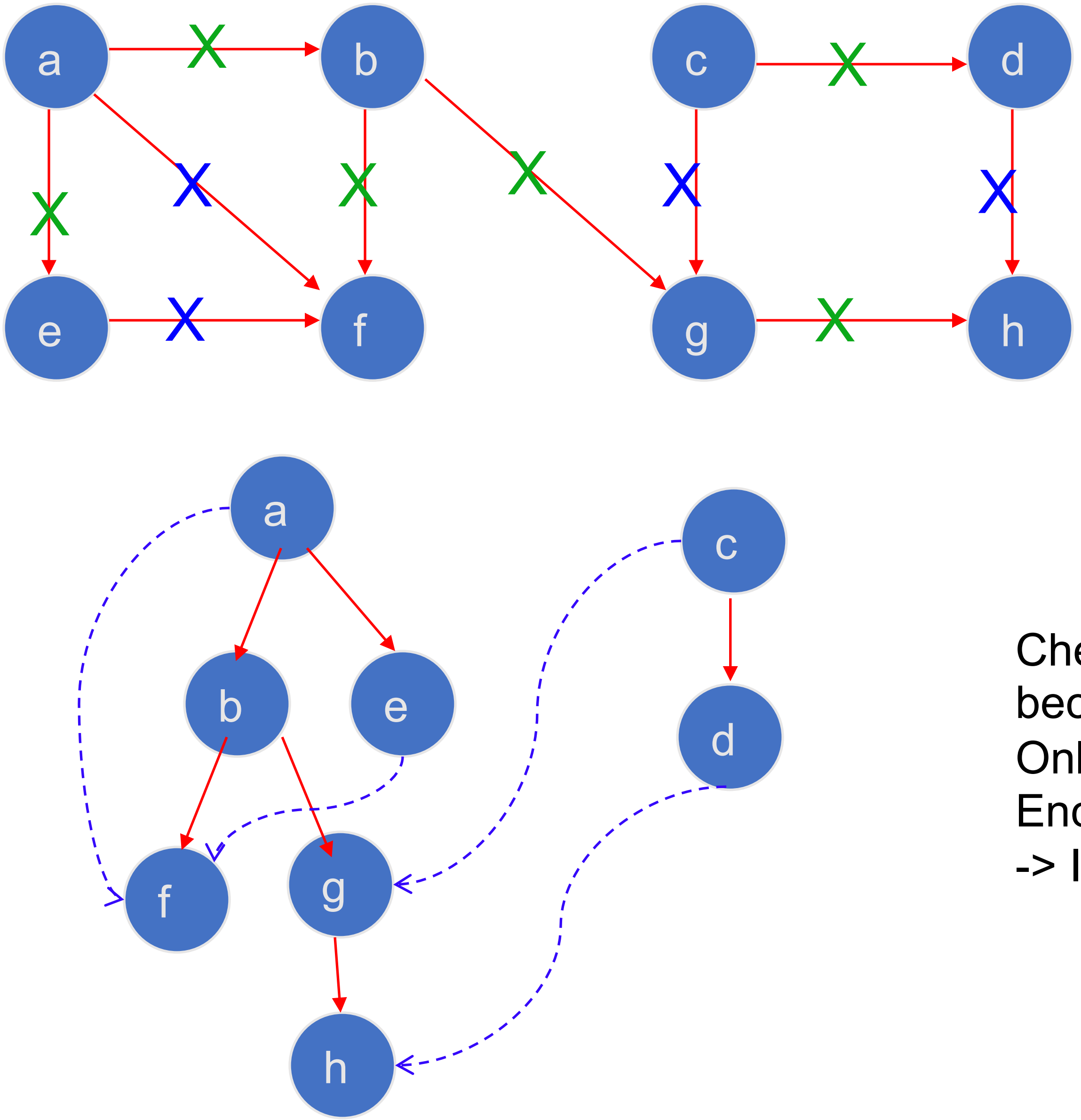
- Perform DFS traversal, noting the order vertices are popped off the traversal stack
- Reverse order solves topological sorting problem
- Check this. Back edges encountered?
  - if yes → NOT a dag! → no solution found.

Example:



Efficiency?

O(n)



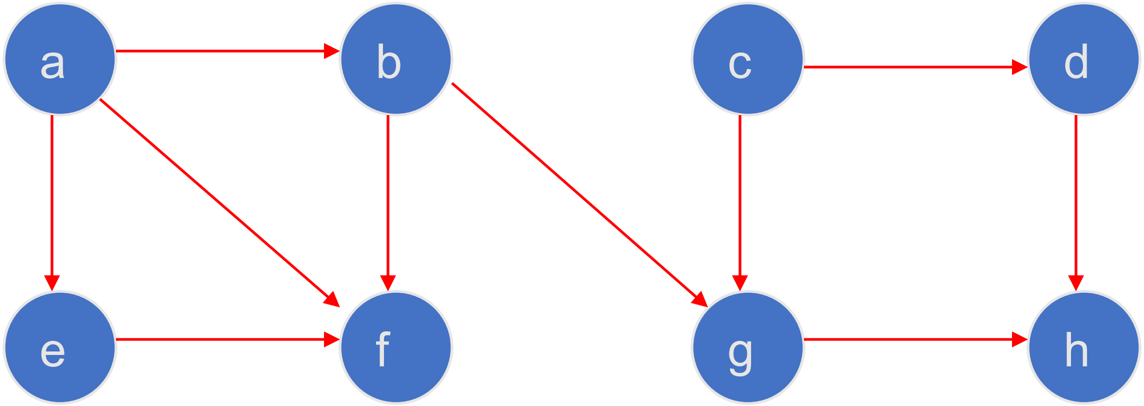
Check the order the vertex becomes dead ->
Only forward edges
Encountered
-> It is a dag

Source Removal Algorithm

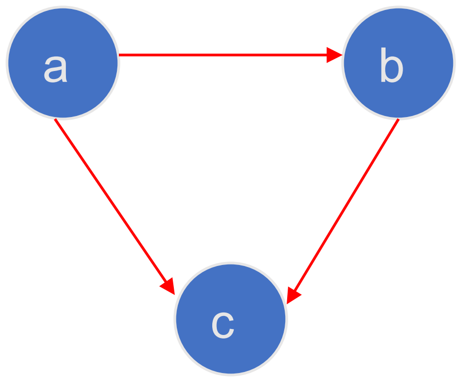
Source removal algorithm

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 (problem is solved) or there is no source among remaining vertices (not a dag)

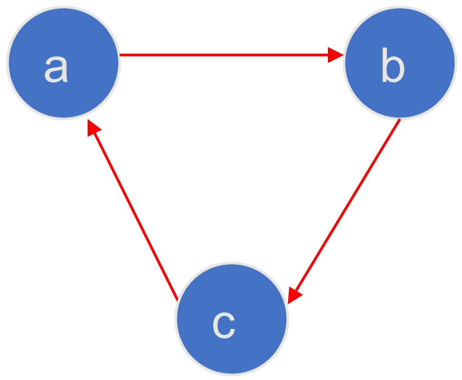
Example:



Efficiency: same as efficiency of the DFS-based algorithm



no vertex is left
-> it is a dag



no source among
remaining vertices
-> it is not a dag

4.3 Generating Permutations

Bottom up algorithm

- If n = 1 return 1;
- Else Given the (n-1)! permutation if 1, 2, …, n-1, insert n into each position of each of them

Example: n=3

start 1

step1 12 21

step2 123 132 312 ; 321 231 213

step3 1234 1243 1423 4123 ; 1324 1342 1432 4132; ….

……

- This approach requires that all the permutations of *1,2,…,(n-1) are calculated already*
  - *Not easy to do! (requires lots of space)*

Permutations of Size n

Generating permutations of size n

- **find all permutations of size n-1 of elements a_1, a_2, .., a_{n-1}**
- **construct permutations of n elements as:**
  - **append a_n to each permutation of size n-1**
  - **for each permutation of size n-1**

    for k from 1 to n-1

    insert a_n in front of a_k
  - **append a_n to the end**

Subsets and Gray Codes

The straight-forward (or bottom up) implementation

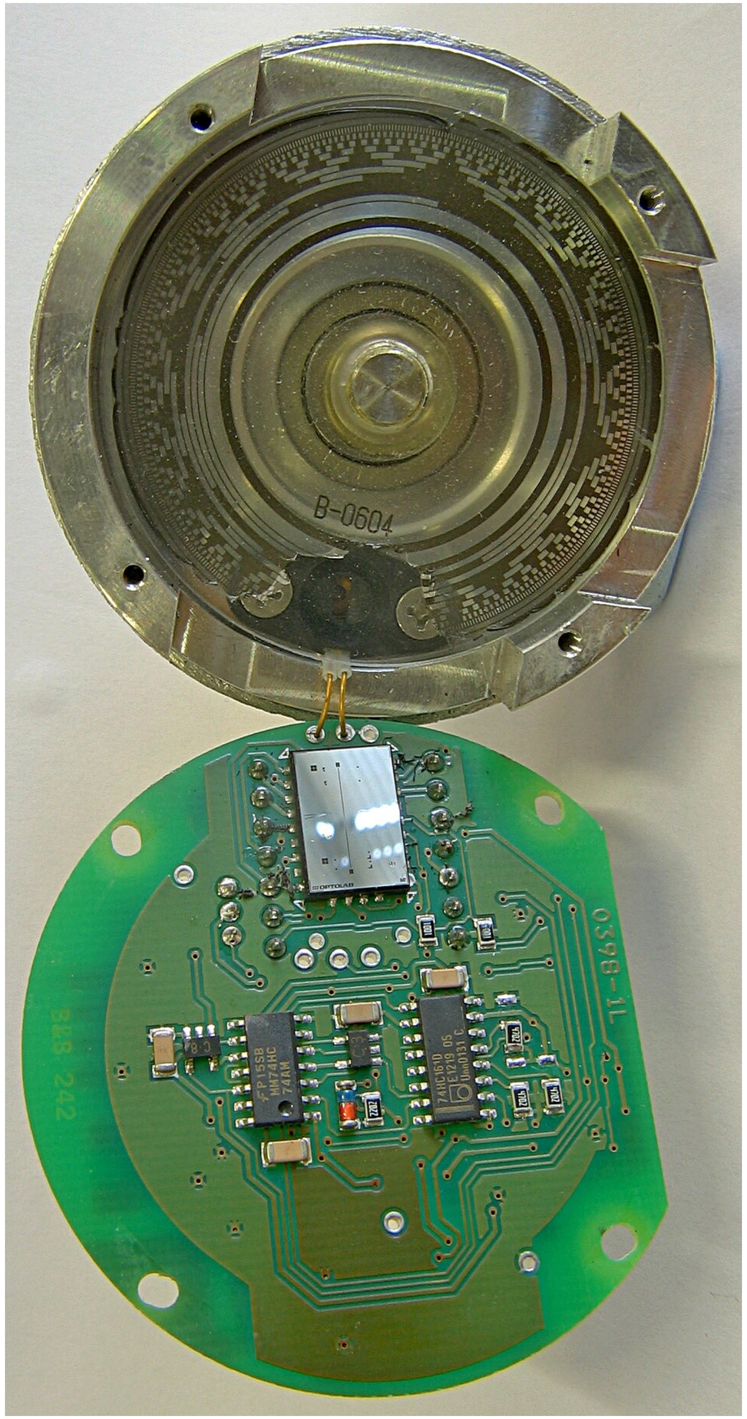
- Let S_{n-1} be the set of all subsets of n-1 elements,
- $S_{n-1} = \{A_1, A_2, \ldots A_m\}$, $m = 2^{n-1}$
- $S_n = \{A_1, A_2, \ldots A_m, A_1 \cup a_n, A_2 \cup a_n, \ldots A_m \cup a_n\}$

Gray Codes:

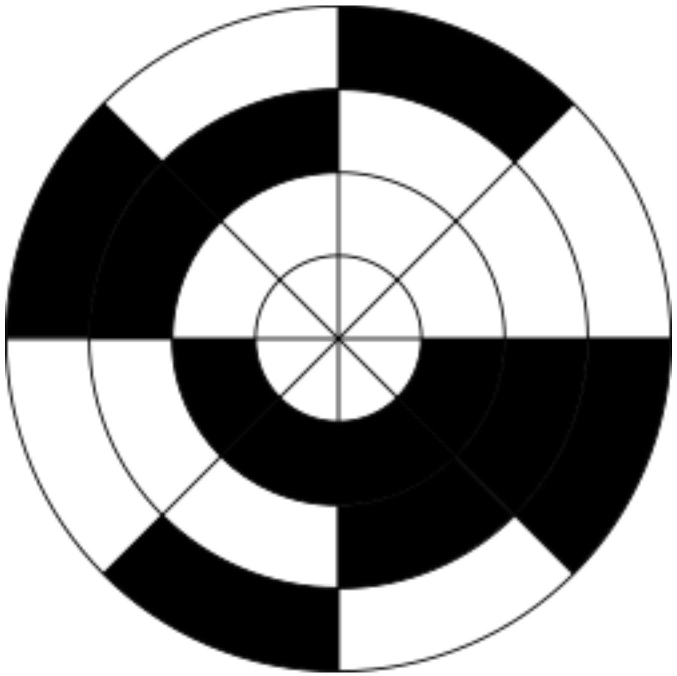
- No need to generate all power sets of smaller sets.
- Frank Gray (1953): a minimal-change algorithm for generating all binary sequences of length n - “binary reflected Gray code”.

Rotary Encoders

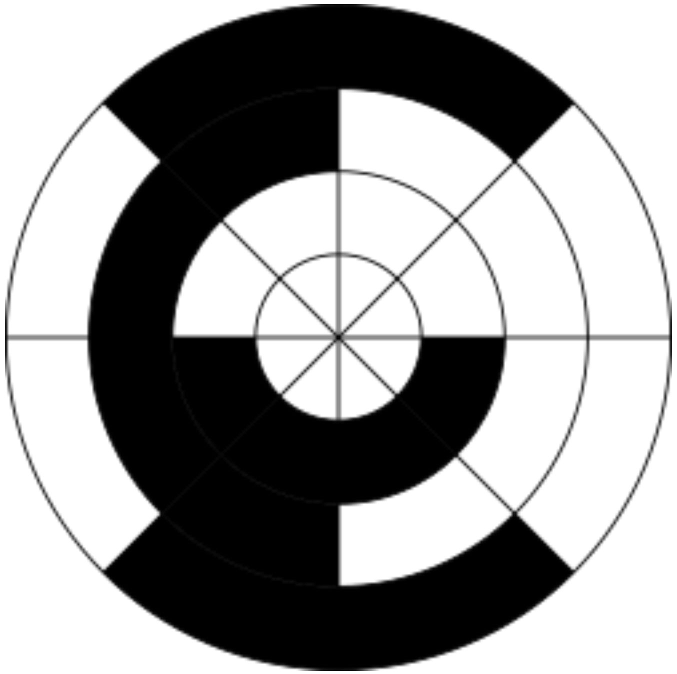
- In robotics, it's often essential to know the *angular position* of a rotating shaft.
- An *encoder* coverts some electrical signal into an estimate of angular position.
- An *optical* encoder shoots light through tiny slits and coverts that to a binary number that is mapped to an angle.
- A *mechanical* encoder does the same thing with small electrically conductive brushes.



Gray Codes



A “natural” binary rotary encoding: each sector is 45 degrees. If the encoders aren’t perfectly aligned there can be catastrophic errors



A “gray code” prevents errors. How? What is different

Generating Subsets

Binary reflected Gray code: minimal-change algorithm for generating 2^n bit strings corresponding to all the subsets of an n-element set where $n > 0$

If n=1 make list L of two bit strings 0 and 1
else
  generate recursively list $L1$ of bit strings of length n-1
  copy list $L1$ in reverse order to get list $L2$
  add **0** in front of each bit string in list $L1$
  add **1** in front of each bit string in list $L2$
  append $L2$ to $L1$ to get L
return L

Subsets and Gray Codes

base	reflect	prepend	reflect	prepend
0	0	00	00	000
1	1	01	01	001
	1	11	11	011
	0	10	10	010
			10	110
			11	111
			01	101
			00	100

Review: Binary Search

- Very efficient algorithm for searching a key in a 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 $l \le r$ 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., $C_w(10^6) = ?$
  - $C_w(10^6) = 20$

Analysis of Binary Search

- Optimal for searching a sorted array

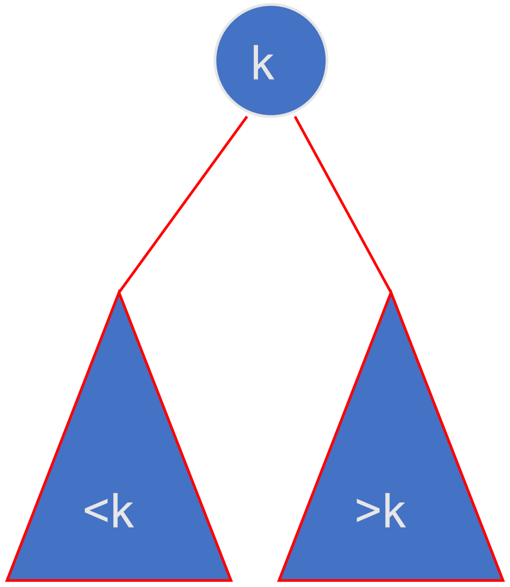
- Limitations: must be a sorted array (not linked list)

- Has a continuous counterpart called *bisection method* for solving equations in one unknown $f(x) = 0$ (see Sec. 12.4 if you like to learn more in this topic, option to you)

Binary Search Tree Algorithms

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

- Searching
  - Efficiency:
    - Best case O(1)
    - Worst case: the height of the tree
    - Average case?
- Insertion of a new key
- Finding the smallest (or the largest) key



Searching in Binary Search Tree

Algorithm *BTS*(x, v)
//Searches for a 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: $C(n) \approx 2 \ln n \approx 1.39 \log_2 n$