Decrease & Conquer: Decrease-by-one

4.1-4.2

Decrease and Conquer technique

Decrease & conquer is a general algorithm design strategy based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem.

This approach is also known as incremental or inductive approach.

Decrease and Conquer technique

Decrease or reduce problem instance to smaller instance of the same problem and extend solution.

Conquer the problem by solving smaller instance of the problem.

Extend solution of smaller instance to obtain solution to original problem.

Implementations of decrease and conquer

This approach can be either implemented as top-down or bottom-up.

Top-down approach : It always leads to the recursive implementation of the problem.

Bottom-up approach : It is usually implemented in iterative way, starting with a solution to the smallest instance of the problem.

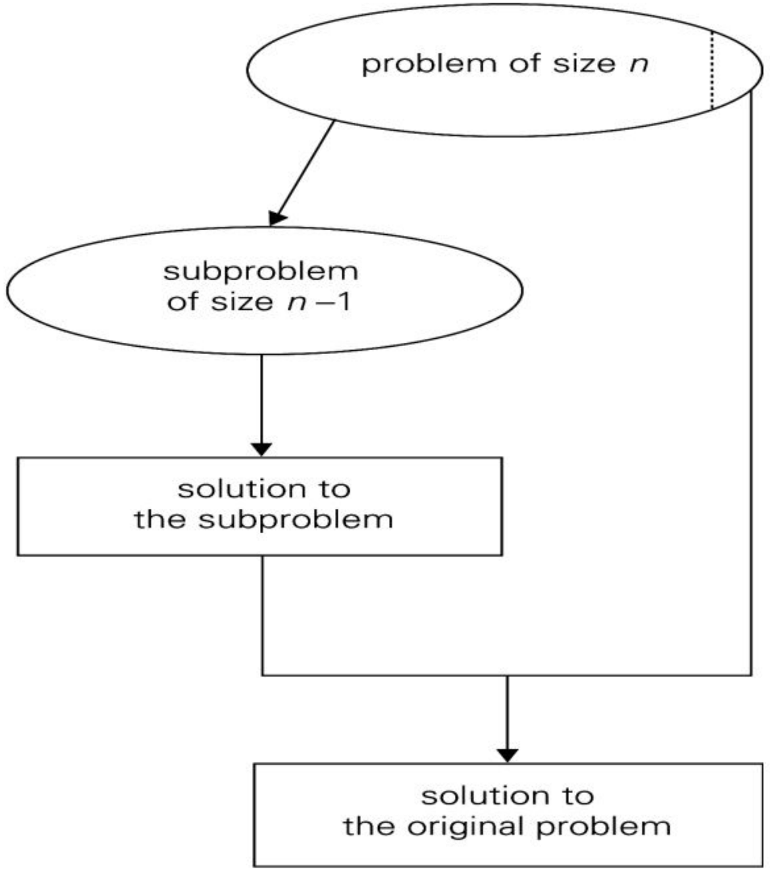
Decrease and Conquer technique

There are three major variations of decrease-and-conquer:

1. Decrease by a constant
    a. Insertion
    b. Topological
2. Decrease by a constant factor
    a. Binary (next lecture)
3. Variable size decrease
    a. Euclids (next lecture)

Decrease by a constant

In this variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one , although other constant size reductions do happen occasionally.



Examples we will cover today include:

1. Insertion sort
2. Topological sorting

Insertion Sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Simple insertion sort pseudo-code

To sort an array of size n in ascending order:

1: Iterate from arr[1] to arr[n] over the array.

2: Compare the current element (key) to its predecessor.

3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Short video explanation

https://www.youtube.com/watch?v=OGzPmgsI-pQ

Example of insertion sort

12, 11, 13, 5, 6

Let us loop for i = 1 (second element of the array) to 4 (last element of the array)

i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12

11, 12, 13, 5, 6

i = 2. 13 will remain at its position as all elements in A[0..I-1] are smaller than 13

11, 12, 13, 5, 6

i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.

5, 11, 12, 13, 6

i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.

5, 6, 11, 12, 13

Let’s look at code examples

- first, meet in groups for 8 minutes to discuss some basics for the code
- then, I’ll send out my code

Complexity of insertion sort

To understand the average and worst case analysis, let’s diagram our code (based on C++):

```
for i = 0 to n
    key = A[i]
    j = i - 1
    while j >= 0 and A[j] > key
        A[j + 1] = A[j]
        j = j - 1
    end while
A[j + 1] = key
end for
```

Complexity of insertion sort

We assume Cost of each i operation as C i where i $\in$ {1,2,3,4,5,6,8} and compute the number of times these are executed. Therefore the Total Cost for one such operation would be the product of Cost of one operation and the number of times it is executed.

We could list them as:

Complexity of insertion sort

Then Total Running Time of Insertion sort

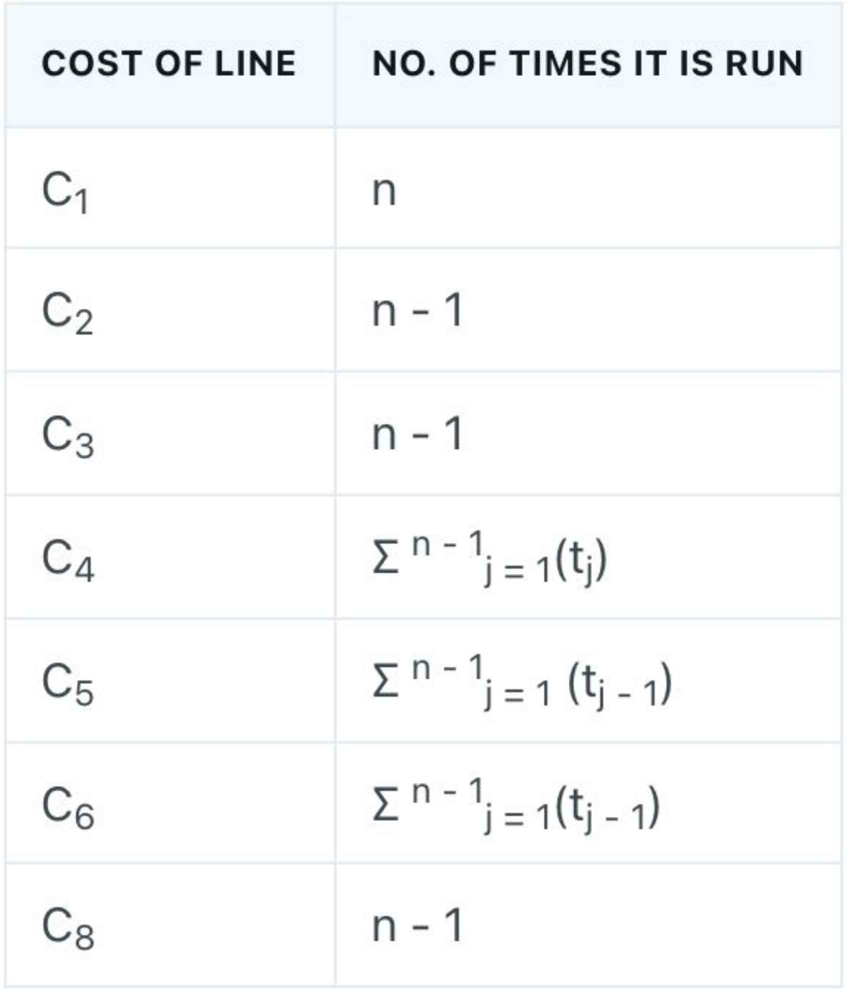
$(T(n)) = C1 * n$

$+ \quad (C2 + C3) * (n - 1)$

$+ \quad \dots$

$+$

$= 1(t_j) + C8 * (n - 1)$



COST OF LINE	NO. OF TIMES IT IS RUN
C_1	n
C_2	$n - 1$
C_3	$n - 1$
C_4	$\Sigma^{n-1}{}_{j=1}(t_j)$
C_5	$\Sigma^{n-1}{}_{j=1}(t_{j-1})$
C_6	$\Sigma^{n-1}{}_{j=1}(t_{j-1})$
C_8	$n - 1$

Complexity of insertion sort

Best Case Analysis:

when the array is already sorted, $t_j = 1$

Therefore,T(n) = C1 * n + (C2 + C3) * (n - 1) + C4 * (n - 1) + (C5 + C6) * (n - 2) + C8 * (n - 1)

which when further simplified has dominating factor of n and gives

T(n) = C * (n) or O(n)

If the input array is already in sorted order, insertion sort compares O(n) elements and performs no swaps. Therefore, in the best case, insertion sort runs in O(n) time.

Complexity of insertion sort

Worst Case Analysis:

when the array is reversely sorted (descending order), $t_j = j$

Therefore,T(n) = C1 * n + (C2 + C3) * (n - 1) + C4 * (n - 1) (n) / 2 + (C5 + C6) * ((n - 1) (n) / 2 - 1) + C8 * (n - 1)

which when further simplified has dominating factor of n^2 and gives

T(n) = C * (n^2) or O(n^2)

Complexity of insertion sort

Worst Case Analysis:

Let's assume that t_j = (j-1)/2 to calculate the average case

Therefore,$T(n) = C_1 * n + (C_2 + C_3) * (n - 1) + C_4/2 * (n - 1) (n) / 2 + (C_5 + C_6)/2 * ((n - 1) (n) / 2 - 1) + C_8 * (n - 1)$

which when further simplified has dominating factor of n^2 and gives

$T(n) = C * (n^2)$ or $O(n^2)$

Can we optimize the run time further?

Searching for the correct position of an element and **Swapping** are two main operations included in the Algorithm.

Can we optimize the run time further?

- We can optimize the searching by using Binary Search, which will improve the searching complexity from O(n) to O(log n) for one element and to n * O(log n) or O(n log n) for n elements.
- But since it will take O(n) for one element to be placed at its correct position, n elements will take n * O(n) or O(n^2) time for being placed at their right places. Hence, the overall complexity remains O(n^2).

Can we optimize the run time further?

- We can optimize the swapping by using Doubly Linked list instead of array, that will improve the complexity of swapping from O(n) to O(1) as we can insert an element in a linked list by changing pointers (without shifting the rest of elements).
- But since the complexity to search remains O(n^2) as we cannot use binary search in linked list. Hence, The overall complexity remains O(n^2).

Can we optimize the run time further?

Therefore, we can conclude that we cannot reduce the worst case time complexity of insertion sort from $O(n^2)$.

Advantages of insertion sort

- Simple and easy to understand implementation
- Efficient for small data
- Chosen over bubble sort and selection sort, although all have worst case time complexity as $O(n^2)$
- Maintains relative order of the input data in case of two equal values (stable)

Applications of insertion sort

- It could be used in sorting small lists.
- It could be used in sorting "almost sorted" lists.
- It could be used to sort smaller sub problem in Quick Sort

Topological Sort

topos means *place* in Greek, I imagine it is because we are sorting elements based on their place in a partial order, rather than on magnitude.

A topological sort is an ordered list of the vertices in a directed acyclic graph such that, if there is a path from v to w in the graph, then v appears before w in the list.

First: what is a directed acyclic graph?

A directed acyclic graph (DAG) is a conceptual representation of a series of activities. The order of the activities is depicted by a graph, which is visually presented as a set of circles, each one representing an activity, some of which are connected by lines, which represent the flow from one activity to another.

Each circle is known as a “vertex” and each line is known as an “edge.”

First: what is a directed acyclic graph?

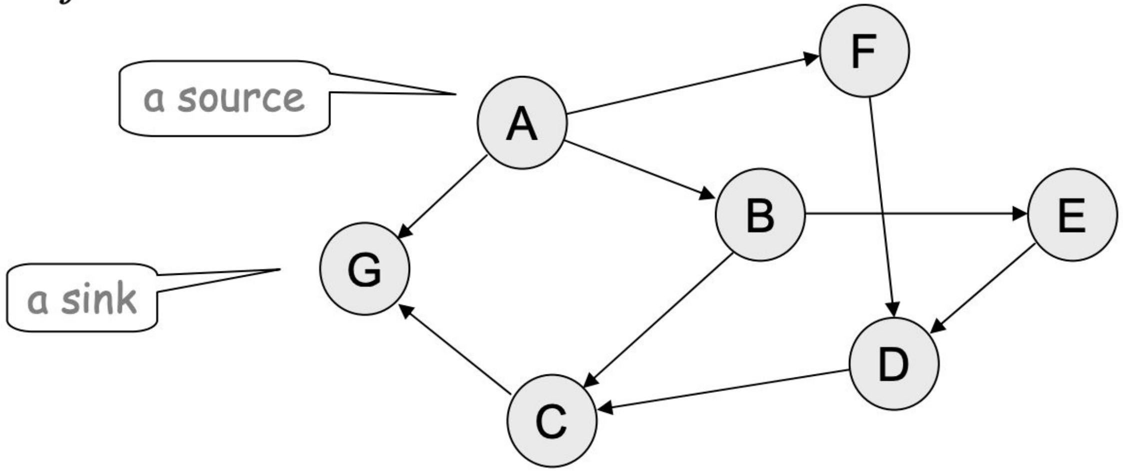
“**Directed**” means that each edge has a defined direction, so each edge necessarily represents a single directional flow from one vertex to another.

“**Acyclic**” means that there are no loops (i.e., “cycles”) in the graph, so that for any given vertex, if you follow an edge that connects that vertex to another, there is no path in the graph to get back to that initial vertex.

First: what is a directed acyclic graph?

In any digraph, we define a vertex v to be a source, if there are no edges leading into v, and a sink if there are no edges leading out of v.

Example of a DAG:



Why are DAGs useful?

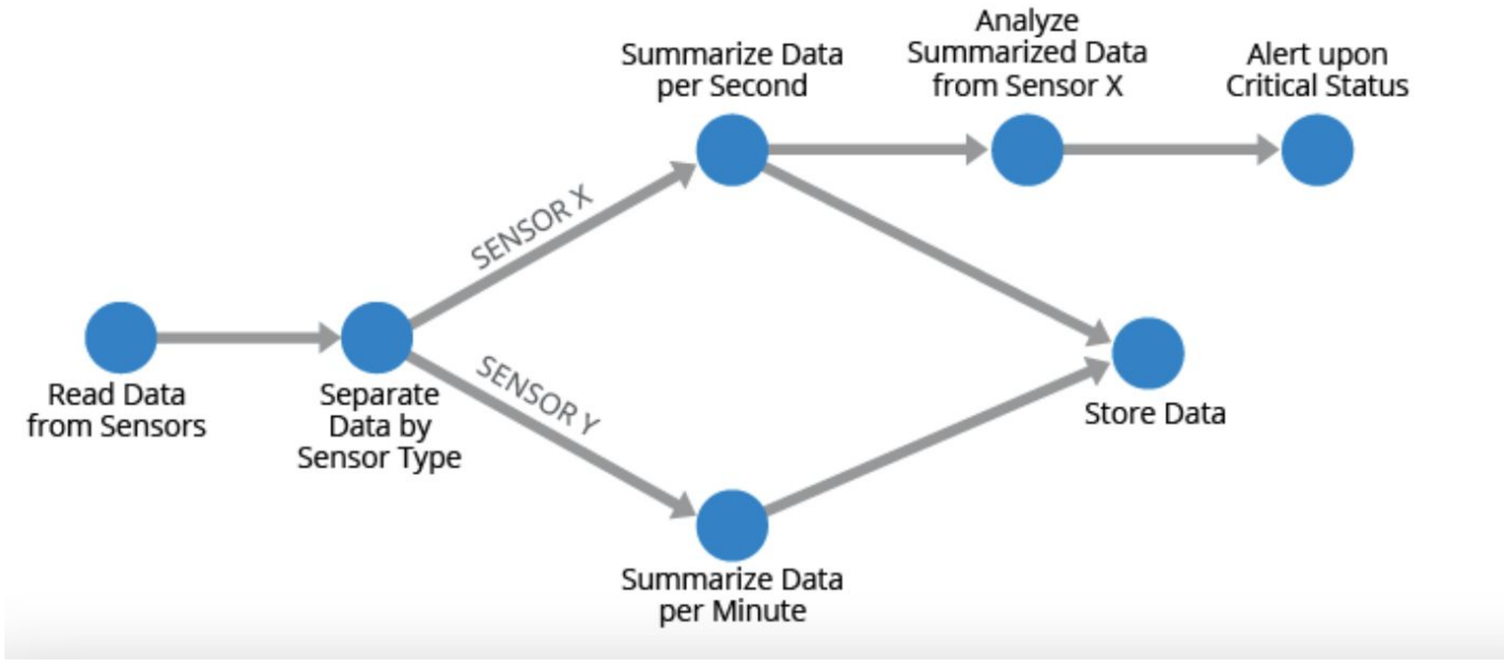
DAGs are useful for representing many different types of flows, including data processing flows. By thinking about large-scale processing flows in terms of DAGs, one can more clearly organize the various steps and the associated order for these jobs.

AKA: data pipeline, process workflow

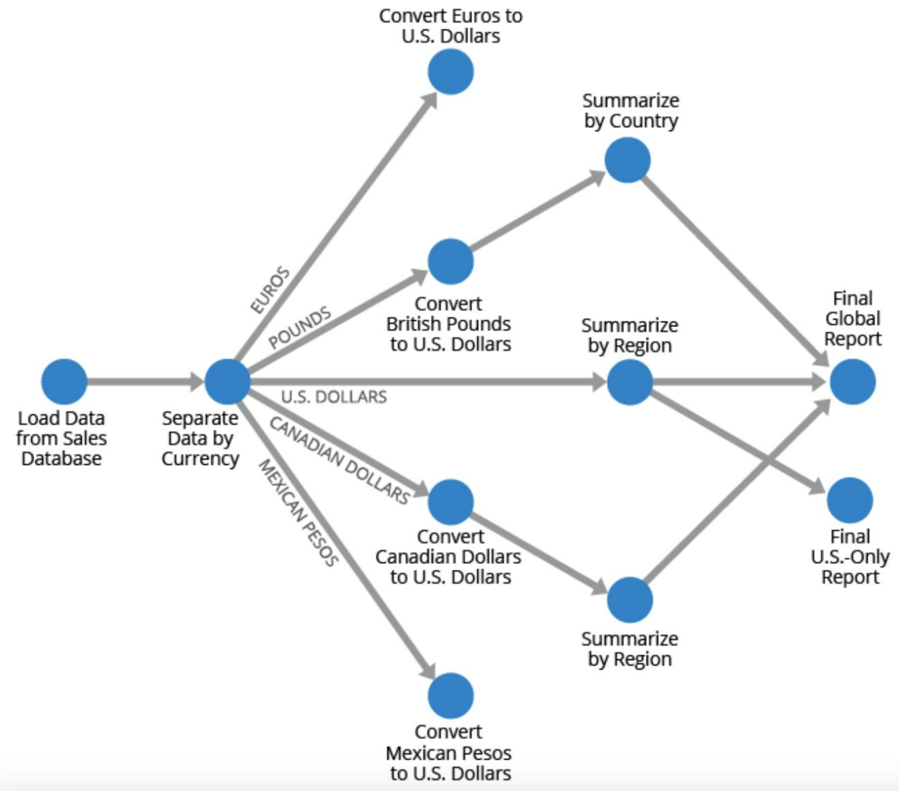
Example of DAG

Sales transaction data might be processed immediately to prepare it for making real-time recommendations to consumers. As part of the processing lifecycle, the data can go through many steps including cleansing (correcting incorrect/invalid data), aggregation (calculating summaries), enrichment (identifying relationships with other relevant data), and transformation (writing the data into a new format).

There can be multiple paths in the flow


Analyze Summarized Data from Sensor X
Summarize Data per Second
Alert upon Critical Status
SENSOR X
Read Data from Sensors
Separate Data by Sensor Type
SENSOR Y
Store Data
Summarize Data per Minute

DAGS also apply to batch processing pipelines


Convert Euros to U.S. Dollars
Summarize by Country
EUROS
POUNDS
Convert British Pounds to U.S. Dollars
Summarize by Region
Final Global Report
Load Data from Sales Database
Separate Data by Currency
U.S. DOLLARS
CANADIAN DOLLARS
MEXICAN PESOS
Convert Canadian Dollars to U.S. Dollars
Final U.S.-Only Report
Summarize by Region
Convert Mexican Pesos to U.S. Dollars

Must be DAG (no cycles)

Topological Sorting for a graph is not possible if the graph is not a DAG.

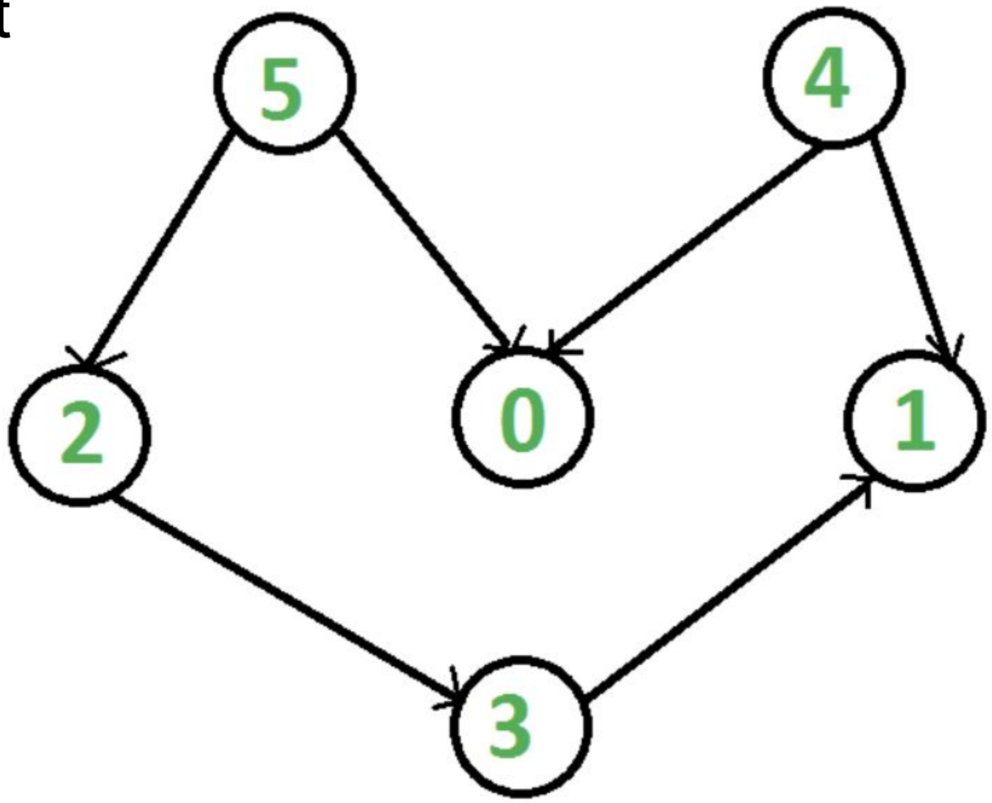
Topological sorting process

In topological sorting, we use a temporary stack. We don’t print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of the stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in the stack.

Short video explanation

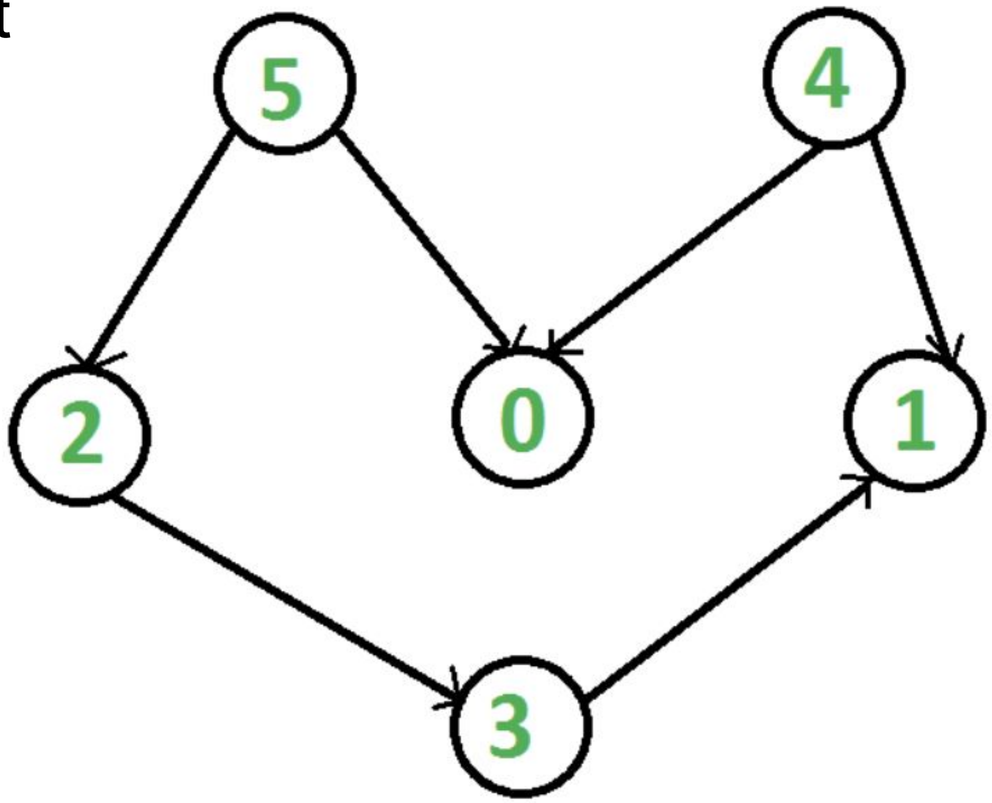
https://www.youtube.com/watch?v=GYmq98CVm2c

Example of topological sort



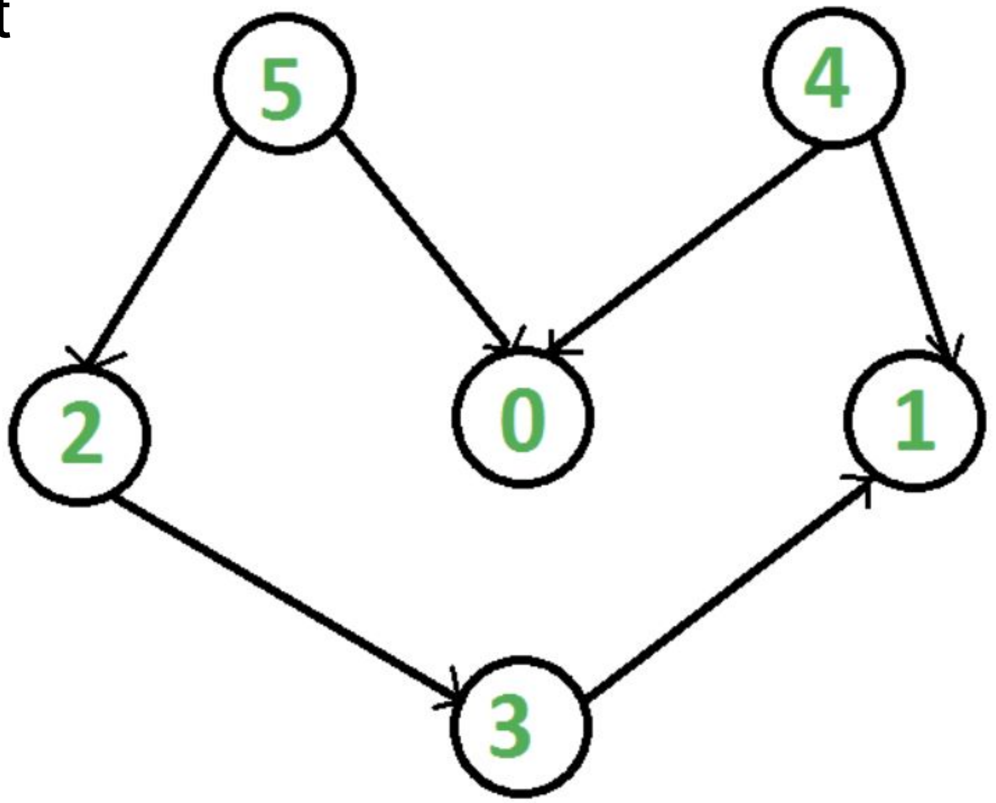
Example of topological sort

1. Start at 5
2. “Visit neighbors” 5 -> 0
3. 0 has no neighbors so push to stack
4. Backtrack to 5
5. “Visit neighbors” 5 -> 2-> 3 -> 1
6. 1 has no neighbors so push to stack
7. Backtrack to 3
8. 3 has no neighbors so push to stack
9. Backtrack to 2
10. 2 has no neighbors to push to stack
11. Backtrack to 5
12. 5 has no neighbors to push to stack
13. Move to 4
14. “Visit neighbors” 4 ->
15. 4 has no neighbors to push to stack



Example of topological sort

452310



Let’s look at code examples

- first, meet in groups for 8 minutes to discuss some basics for the code
- then, I’ll send out my code

Complexity of topological sort

The topological sort has a O(V+E) time complexity.

Article with possible reduced complexity

https://link.springer.com/chapter/10.1007/978-3-030-68154-8_38

Applications of topological sort

- Scheduling jobs from the given dependencies among jobs.
- Instruction scheduling
- Ordering of formula cell evaluation when recomputing formula values in spreadsheets,
- Logic synthesis,
- Determining the order of compilation tasks to perform in make files,
- Data serialization, and
- Resolving symbol dependencies in linkers