AADS Lecture Notes

Lecture 1 (Union Find) (Chapters: 1, 8.1-8.5, 8.7)

- An algorithm is a method for solving a problem
- A data structure is a method to store information
- Union Find
  - Two Operations (Union and Connected)
    - Union Find
      - Each connected component shares the same id.
      - Connected operation is constant
    - Quick Union
      - Each connected component is seen as a tree. If nodes share a root they are connected.
      - Connected now requires traversing the tree with the root function.
      - Weighted Quick Union
        - Make the smaller tree the child of the larger tree. This aims to better balance the tree.
        - Connected requires traversing the tree with the root function.
          - Path Compression
            - Move subtrees up the tree while we are looking for the root.

Lecture 2 (Algorithm Analysis) (Chapters: 2)

- Growth
  - Linear, Exponential, Logarithmic

$$a \cdot x^b$$

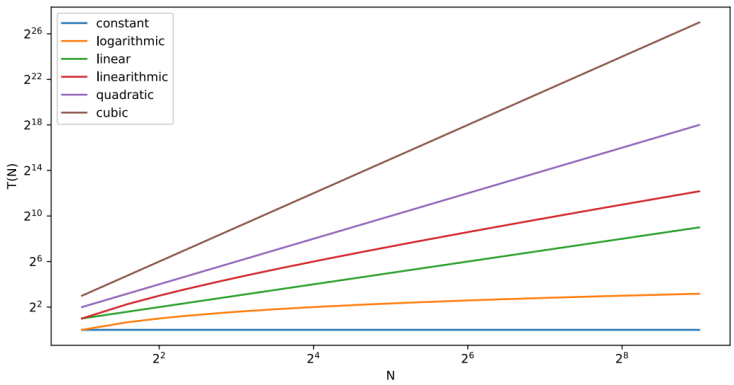
  - Growth in log-log space: $a = 2^c$
    - Compute the line with-

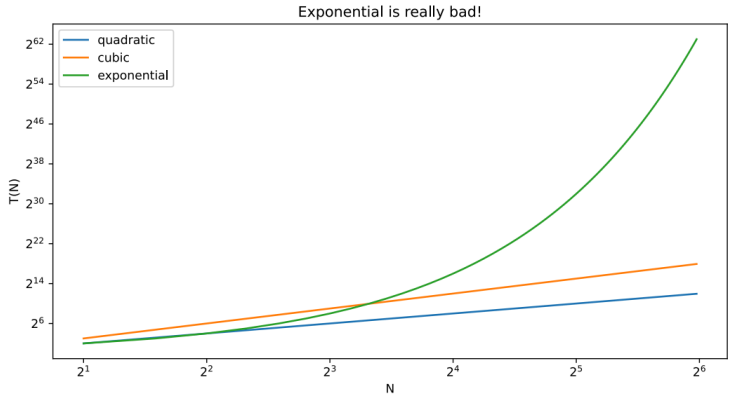
        - $b = \frac{\log_2 y_1 - \log_2 y_0}{\log_2 x_1 - \log_2 x_0}$
        - $c = \log_2 \text{time}(x) - b \cdot \log_2 x$
        - $2^c * x^b$
- K-sum (How many sequences of k numbers in this list sum to 0?)
    - Normal 2-sum: quadratic
        - 2-sum with caching
            - Store numbers we have seen already in cache.
            - If we have seen the current number as negative before it is a match.
            - Increases memory usage
        - 2-sum with sorting
            - Use a sorted list and 2 pointers one and the front and one at the end.
            - If too small pick larger number in front
            - If too large pick a smaller number in the end
    - Normal 3-sum: cubic
        - 3-sum with 2-sum pointers: quadratic
            - Modify 2-sum to take a target value and then iterate over teh list and use 2-sum to look for two numbers that sum to the target number inverse.
        - 3-sum with pointers plus cache and 2-sum with pointers
- Math Models
    - We can estimate run times by counting operations
        -
```
 1                              # freq
 2  count = 0                   # 1
 3  i = 0                       # 1
 4  while i < N:                # N + 1
 5    j = i+1                   # N
 6    while j < N:              # 0.5 * (N+2) * (N+1)
 7      if a[i] + a[j] == 0:    # N * (N-1) 2 * (0+1+2+...+(N-1))
 8        count += 1            # 0 to 0.5 * N * (N-1)
 9      j += 1                  # 0.5 * N * (N-1)
10    i += 1                    # N
```
    - We can use a cost model
        - Use some basic operation as a proxy for running time
            - Too much effort and guessing to determine exactly how many times each operation is performed
    - Tilde notation
        - We estimate runtime or memory use as a function of input size N
        - If we add things together, we will get a number of terms
            - As N grows, the lower order terms are negligible
            - And if N is small, we do not care
        - So, $N^3 + 5 * N^2 + 100 * N + 10987 \sim N^3$

- $f(N) \sim g(N)$ means $\lim_{N \to \infty} \frac{f(N)}{g(N)} = 1$
    - This means as N approaches infinity the relation of f(N) to g(N) approaches 1.
- Definitions from the book
    - Definition 1
        - https://people.engr.tamu.edu/djimenez/ut/utsa/cs3343/lecture3.html
        - $T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) <= cf(N)$ when N >= n_0
            - Consider 1000N and N^2. There are values for N where 1000 * N is larger, but N^2 ***grows*** faster
            - There is some point n_0 after which N^2 is always larger than 1000N
        - If $T(N) = 1000N$ and $f(N) = N^2$, $T(N) <= cf(N)$ when:
        - c=1 and n_0=1000, c=100 and n_0=10
    - Definition 2
        - $T(N) = \Omega(g(N))$ if there are positive constants c and n_0 such that $T(N) >= cg(N)$ when $N >= n_0$
            - A function T(N) is bounded from above by another function g(N), denoted by an upper limit. T(N) cannot grow faster than a constant multiple of g(N) for sufficiently large N.
        - $T(N) = \Theta(h(N))$ if and only if
            - $T(N) = \Theta(h(N))$ and
            - $T(N) = \Omega(h(N))$
                - A function T(N) has the same order of growth as h(N). In other words, T(N) grows at a similar rate as h(N) and also serves as an upper bound for h(N).
- Tighter bounds
    - If f(N) = 2N
        - f(N) = O(N^2) is correct because N^2 is an upper bound for 2N. It means that f(N) doesn't grow faster than a quadratic function.
        - f(N) = O(N^3) is also correct for the same reason. N^3 is an upper bound, and f(N) grows more slowly than cubic.
        - f(N) = O(N) is a valid upper bound and a **more precise** one than the previous two. Since f(N) is linear (2N), it doesn't grow faster than O(N). In this case, O(N) is the tightest, most specific upper bound for f(N).
- Look Through This Lecture Starting At Slide 135

- Functions To Describe Growth

order	name	description	T(2N)/T(N)
1	constant	statement	1
log N	logarithmic	divide in half	~ 1
N	linear	loop	2
N log N	linearithmic	divide and conquer	~ 2
N^2	quadratic	double loop	4
N^3	cubic	triple loop	8





- Types of analyses
  - Best Case, lower bound on cost
    - "Easiest" Input
    - Provides a goal for all inputs
  - Worst case, upper bound on cost
    - "Most difficult" input
    - Provides a guarantee for all inputs
    - The **worst case bound is the best** analytical result known
  - Average case, expected cost for random input
    - Provides a way to predict performance
  - Commonly Used notations

notation	provides	example
Big Theta	asymptotic order of growth	$\Theta(N^2)$
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$

❖ Lecture 3 (Lists, Stacks, and Queues) (Chapters: 3)

- ➢ An ADT is a theoretical (mathematical) model
  - ■ It defines the behavior of a data type, not its implementation
    - They type is completely opaque from outside of its operations
  - ■ The behavior is defined from the user's point of view, i.e., as a set of operations.
  - ■ It consists of a set of values or elements, the domain
    - If finite, this can be specified via enumeration
    - If not, via some rule describing the elements
  - ■ And a set of operations
    - Syntax, names and values
    - Semantics, functionality / behavior
  - ■ Example: Integer is and ADT
    - The domain is all integers
    - The operations include
      - ◆ +,-,*,/ etc
  - ■ Semantics
    - The semantics can be specified in a few different ways
      - ◆ Text
      - ◆ Algebraic specification
      - ◆ Operational specification
    - Example Algebraic Specification

      Syntax for a Stack that can hold integers

      ◆

      $$\begin{array}{lcll} \text{newstack} & : & & \rightarrow \text{STACK} \\ \text{push} & : & \text{STACK} \times \text{INT} & \rightarrow \text{STACK} \\ \text{pop} & : & \text{STACK} & \rightarrow \text{STACK} \\ \text{top} & : & \text{STACK} & \rightarrow \text{INT} \cup \{\text{Error}\} \\ \text{empty} & : & \text{STACK} & \rightarrow \text{BOOLEAN} \end{array}$$

      ◆

      $$\begin{aligned} \text{empty(newstack)} &= \text{True} \\ \text{empty(push(s, i))} &= \text{False} \\ \text{pop(newstack)} &= \text{new} \\ \text{pop(push(s, i))} &= \text{s} \\ \text{top(new)} &= \text{Error} \\ \text{top(push(s, i))} &= \text{i} \end{aligned}$$

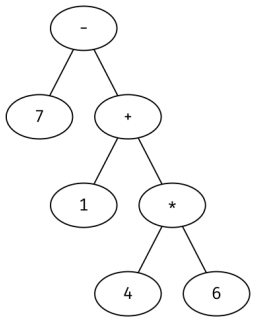
      ◆ where $s \in STACK$ and $i \in INT$.
  - ■ Abstract vs Concrete (DT)
    - An ADT is a model, not an implementation

- It is a theoretical tool, not a language construct
- The model can be implemented in multiple ways
  - Sometimes with restrictions: for example size of signed ints can overflow because of a restriction

➢ Lists (a finite ordered, not sorted, sequence of elements)
- Array/memory based list
  - Double the size of the list and copy over when we hit the limit.
- Linked List
- Doubly Linked List

➢ Stack (can be used to handle function calls when running programs)
- A stack is an ADT where values are inserted and removed from the top
  - Similar to a stack of items in the real world
- LIFO
- Stack Calculator

1. Pop operator `Op.MUL`
2. Pop 4 and 6
3. Push 24 (4 * 6)
4. Pop operator `Op.ADD`
5. Pop 24 and 1
6. Push 25 (24 + 1)
7. Pop operator `Op.SUB`
8. pop 25 and 7
9. push 18 (25 - 7)
10. pop 18 (Result, no more operators)

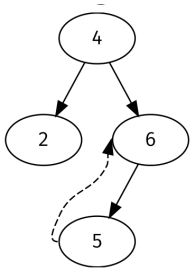


➢ Queue
- A queue is an ADT where values are inserted and removed from each end
  - Similar to a queue/waiting line in the real world
- FIFO
- Two main operations enqueue and dequeue (insert and remove)
- If we insert and delete from an array backed queue and we simply double the array size when needed this can lead to wasted space.
  - We can fix this with circular queues
    - The back and front will move as we enqueue and dequeue
    - A lot of spaces will potentially be wasted
    - So, let's reclaim it bny wrapping around when we hit the end. If the positions are free…
    - If not we can extend the queue by doubling and copying it over
    - Example:
      - Assume we have a queue of size 10
      - The back is at index 9 and the front is at index 7
      - We could grow, but there is no need
      - Instead, we inset at (9 + 1) % 10

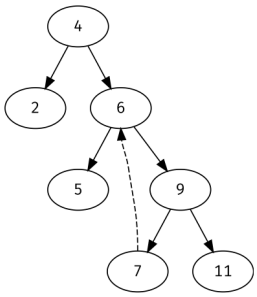
        - This wraps around the end and we are back at index 0
- Set
    - A set is an unordered collection of unique elements
    - The main operation is set membership
        - As well as a way to combine sets, e.g., union
        - Intersection
        - Difference
        - Add, delete, contains
    - Mutable and immutable
- Bags
    - Checking for duplicates is expensive
    - So, a bag is set that allows duplicate values
    - Makes insert easier
        - But some of the other operations a bit more complicated
        - E.g., delete must now delete all instances of a value
    - Can save time if many adds
        - At the cost of space

❖ Lecture 4 (Trees) (Chapters: 4.1-4.6)

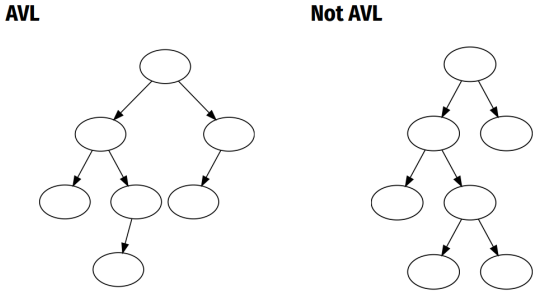
- The Tree ADT
    - A tree is a collection of nodes
    - Each parent can have 0 or more children
    - The root of each subtree is a child of the tree root, and the tree root is the parent of each subtree
    - Nodes with the same parent are siblings
    - The length of a path is the number of edges it contains
    - Depth of a node n is the length of the path from the root to n
    - Height is the longest path from node n to a leaf
        - Leaves have height 0
    - Ancestor / Descendents
- Binary Trees
    - Each parent has at most 2 children
        - Smaller to left larger to right
    - Average tree has height $\Theta(\sqrt{n})$
    - A “full” tree has height log(n) - 1
    - A “degenerate” tree has height n - 1
    - Deleting
        - Assume we want to delete 6
            - If the node has one child we just lift it



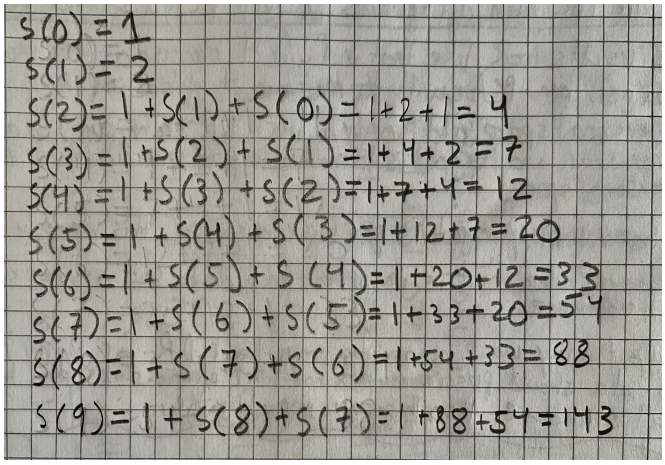
- Harder when it has two children
- Replace the node with the smallest value in the right subtree



- AVL-tree (Adelson-Velskii and Landis)

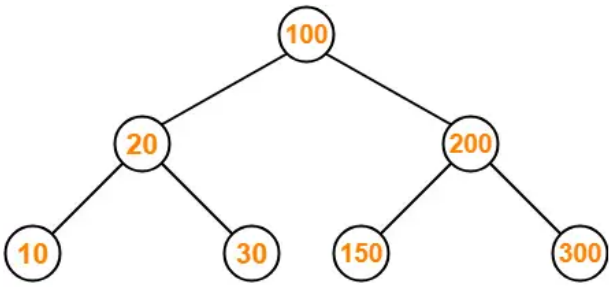


- This is a BST with a *balance condition*
  - AVL Tree - Single Rotation
  - AVL Tree - Double Rotation
  - Each node should have left and right subtrees of heights within 1 of each other
    - This gives a height of about $1.44 * log(N + 2) - 1.328$
      - More than $\log_2$, but not that much
    - Minimum nodes at a height
      - $S(h) = S(h - 1) + s(h - 2) + 1$
      - So a tree with height 9 has at least 143 nodes

- ➢ 

- Inserting can break the balance so we balance after insert
  - LL, LR, RL, RR (LL and RR are easier)
    - ➢ Double right means
      - Rotate right child left
      - Rotate self right
    - ➢ Double left means
      - Rotate right child right
      - Rotate self left

- Walking
  - Pre-order
    - ➢ 1. Visit the current (root) node.
    - ➢ 2. Recursively traverse the left subtree in pre-order.
    - ➢ 3. Recursively traverse the right subtree in pre-order.
  - Post-order
    - ➢ 1. Recursively traverse the left subtree in post-order.
    - ➢ 2. Recursively traverse the right subtree in post-order.
    - ➢ 3. Visit the current (root) node.
  - Example:



Binary Search Tree

- Preorder- 100 , 20 , 10 , 30 , 200 , 150 , 300
- Postorder- 10 , 30 , 20 , 150 , 300 , 200 , 100
- Inorder- 10 , 20 , 30 , 100 , 150 , 200 , 300

- Splay Trees
  - Many applications have “data locality”
    - A node is accessed multiple times within a reasonable timeframe
  - Splay trees push a node to the root after it is accessed
  - Uses a series of rotations from the AVL trees
  - Can also balance the tree
  - Amortized cost- describes the average cost of performing a sequence of operations over time, rather than the cost of each individual operation.
    - Splay trees guarantee that m consecutive operations is (O m log n)
    - A single operation can still be Θ(n), so the bound is not O(log n)
    - This is called the amortized running time
      - If m operations are O(m * f(n))
      - The amortized cost is O(f(n))
    - A zig-zig means that the same rotation is performed twice
      - LL or RR
    - A zig-zag means that a rotation followed by the mirror
      - LR or RL
    - Some use zig for one and zag for the other and have four combinations
      - We can move any node to the root by combining zig , zig-zig, and zig-zag
        - We do this each time we search for a node
        - This will ensure that nodes that we have searched for will be closer to the root.

❖ Lecture 5 (Hashing) (Chapters: 5.1-5.5)

- We use the key to map to an element (we can use %)
- Hashing is a one way function
- There is no order

- ➢ Try to avoid repetition and “round” numbers
  - It is generally a bad idea to use sizes of even 10s
  - Or powers of two
  - Use prime numbers to break patterns
  - Preferably close to powers of two
- ➢ Use as much of the key as possible
  - More bits means more variation
- ➢ Separate chaining
  - In separate chaining, each slot or bucket in the hash table contains a data structure, typically a linked list or sometimes another hash table. When a collision occurs (i.e., multiple keys map to the same hash index), the new key-value pair is inserted into the corresponding data structure at that index. This data structure effectively "chains" together all the key-value pairs that hash to the same index.
  - Randomly toss values into bins
    - We can expect two balls in the same bin after $\sqrt{\pi(m/2)}$ tosses
    - Every bin has >= 1 balls after (m ln m) tosses
    - After m tosses, the most loaded bin has Θ($log\ (m)/log\ log(m)$)
    - A good number of buckets(m) is about n / 5
      - ◆ Then access can be said to be O(1) or constant
- ➢ Linear probing
  - In linear probing, when a collision happens, the algorithm searches for the next available slot by moving linearly through the table until it finds an empty slot. This process continues until an empty slot is found, at which point the key-value pair is inserted into that slot.

  - Knuth’s parking problem
    - Imagine a parking lot with 'n' spaces, where 'n' is a positive integer. The parking lot is empty, and 'n' cars arrive sequentially. Each car driver chooses a parking space at random and parks their car. However, if the chosen parking space is already occupied, the driver will continue to the next available space until an empty one is found.
    - m=num parking spaces
      - ◆ What is the average displacement?
        - ➢ With m / 2 cars, about 3/2
        - ➢ With m cars, about $\sqrt{\pi(m/8)}$
      - ◆ $1/2(1 + (1 + (1/1 - a))$

➢ This formula is related to the expected number of probes required for a successful search in a hash table

◆ $1/2(1 + (1 + (1/1 - a)^2))$

➢ This formula is for the average number of probes when performing an unsuccessful search.

◆ Rule of thumb a = n / m ~ 1/2

➢ Probes for a hit is about 3/2

➢ Probes for miss/insert is about 5/2

- More can be done with quadratic probing, double hashing, etc.

■ Summary

- We can manage collisions with separate chaining or linear probing
- Within constant time on average
- But logarithmic time worst case
- Assuming uniform hashing

❖ Lecture 6 (Priority Queue) (Chapters: 6.1-6.5, 7.5)

➢ Priority Queue

■ Remove the largest or smallest item

■ Remove max/min method and add max/min functions

■ The list thus must be ordered

- Either insert or remove must be O(N) and the other O(1)

■ We can use BSTs to implement priority queues making add or remove to be O(log n) and the other O(1)

➢ Binary heaps

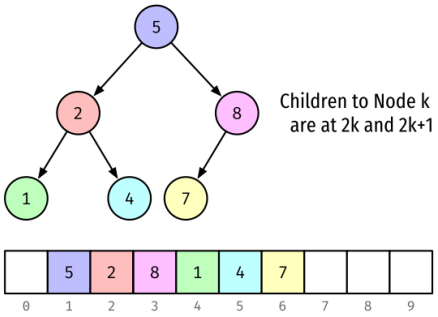
■ This is an almost complete binary tree that satisfies the heap property

■ We can store this binary tree as an array

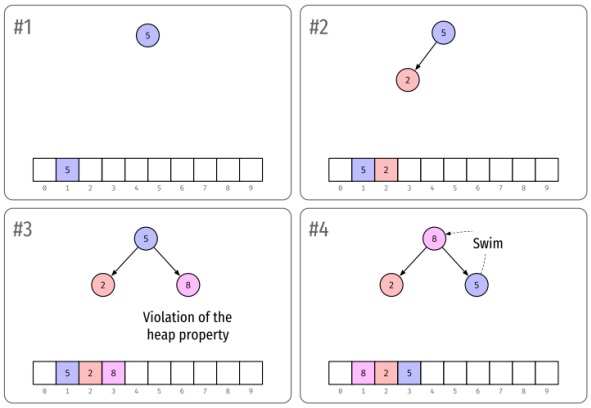
- Removes the need for links

■ The tree is stored level by level

■ We can use 1-based indexing

■ 

■ Parent key is no smaller or larger than children keys depending on min heap or max heap

- Swim
  - If a child's key becomes larger / smaller than a parents key
    - It needs to swim / bubble up
  - Exchange key in child with key in parent until the heap property is restored
  - 

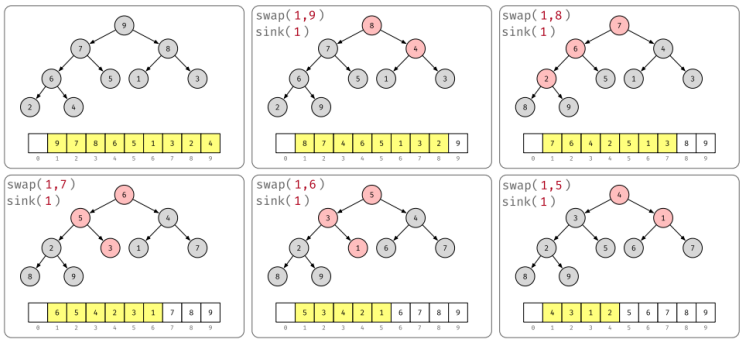
- Sink
  - Opposite of swim, if a parent's key becomes smaller/larger than one or both of the children's
    - It should sink or bubble down
  - Exchange key in parent with key in larger child
    - Until heap property is restored

➢ Analysis
- Sink and swim are expensive
- Swim at most 1+logN compares
- Sink at most 2 * logN compares
- Insert and delMax are O(logN)

➢ d-Heaps
- The tree doesn't have to be binary, the more children the shallower
- Find becomes cheaper
  - Still O(logN) just that $\log_3 1000$ is smaller than log1000
- And delete becomes more expensive (O(d logbased N))

➢ Sorting
- We can fill in a heap then use delMax to populate a list
  - However this basic way requires 2n space so we can do it in place instead
- Heap sort
  - Heap Sort | GeeksforGeeks
  - Build a max heap
  - Remove the max n times, but leave it in the array

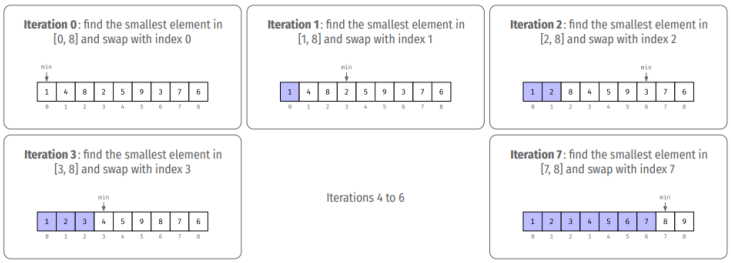
Sort down (remove max)



-
- Analysis
  - 2N log N best and average case
  - O(N log N)

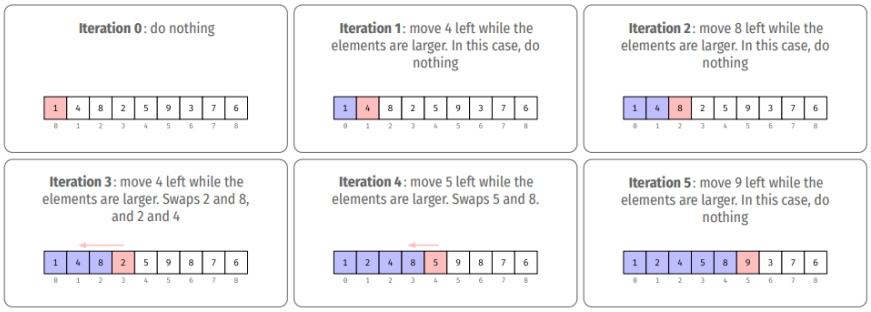
❖ Lecture 7 (Sorting) (Chapters: 7.1-7.11)

- ➢ We are using comparison-based sorting
- ➢ Total order
  - A total order is a binary relation <= that satisfies:
    - Antisymmetry: if v <= w and w <= v, then v = w
    - Transitivity: if v <= w and w <= x, then v <= x
    - Totality: either v <= w or w <= v or both
  - Standard order for example natural or real numbers
- ➢ Sorting Terms
  - In Place
  - Stable: Elements with the same value maintains their relative order
- ➢ Selection Sort (In place and unstable)
  - Simple idea: in iteration i, find the index of the smallest remaining element and swap the element at index i and the smallest element

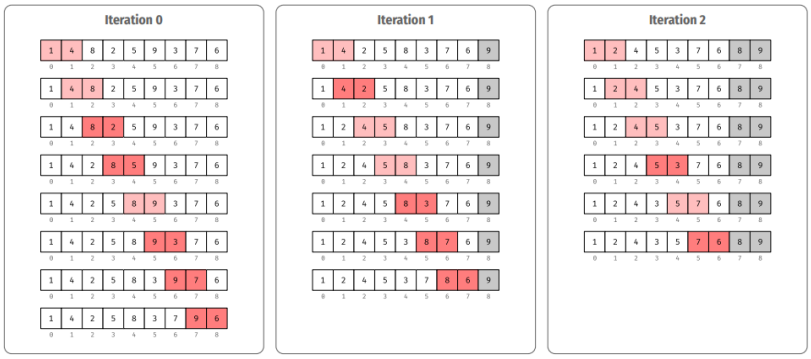


  -
  - This is insensitive to input, O(n^2) whether sorted or completely random
  - Minimal data movement

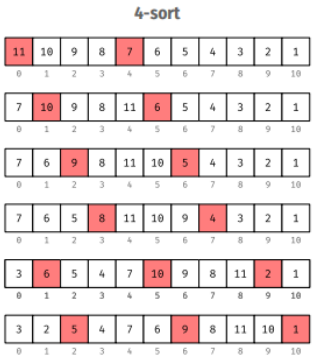
- Insert Sort (In-place and stable)
  - In iteration i, swap the value at index i with each larger entry to its left
  - So, move the value at index i to the correct place

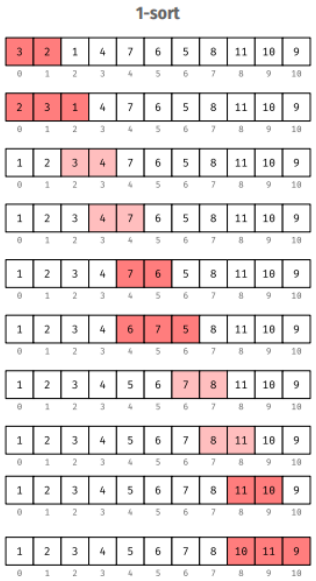


  - Depends on input
    - If sorted, n-1 compares and 0 exchanges
    - If descending order .5 * n^2 compares and exchanges
    - Average case .25 * n^2
  - Still O(n^2), but runs in linear time if partially sorted
- Bubble sort (in place and stable)



  - Similar to insert sort
    - Depends on input, if almost sorted linear
  - So, O(n^2)
- Shell sort (In palace and not stable)
  - Shell Sort | GeeksforGeeks
  - Move elements more than one position at a time
  - H-sorting
  - If h is 4
    - Check lst[h] < lst[h+4]
  - Shellsort
    - H-sort the array with decreasing values of h
      - 13 sort, 4 sort, 1 sort
  - We use insertion sort with stride h
  - Big increments, small subarray
  - Small increments, nearly in order

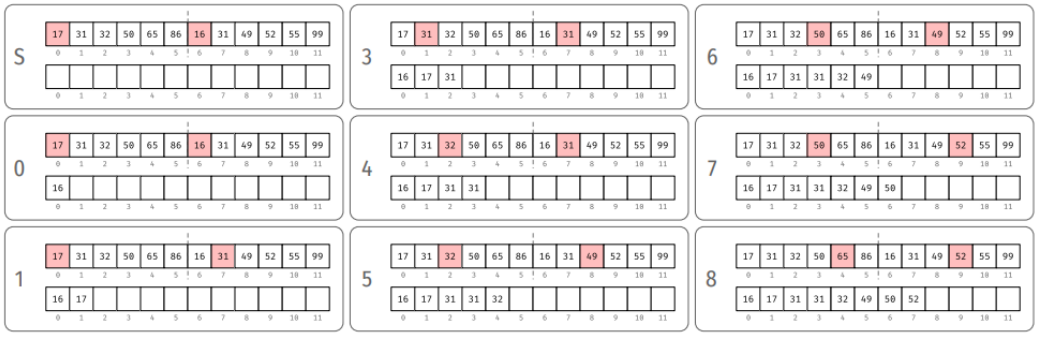




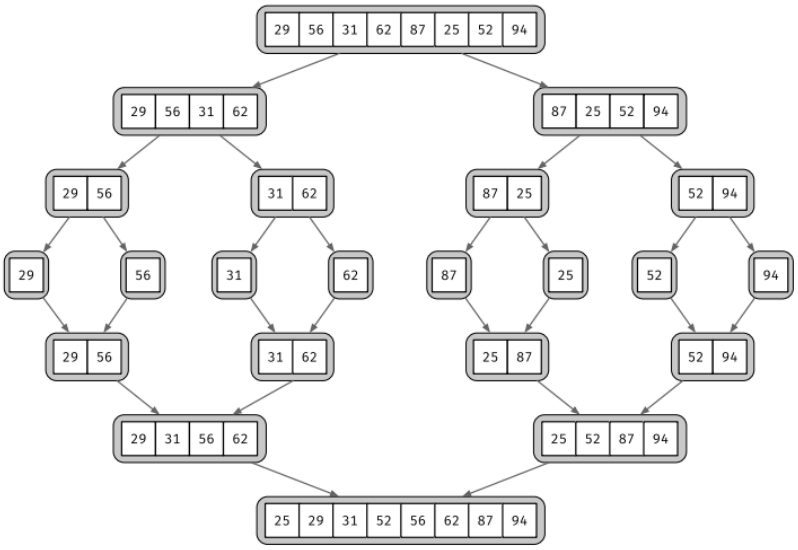
-
- Which Sequeue of h?
  - Any should work, but some are better than others
  - Powers of two are bard
  - 3x-1 is ok
    - Performs well and is easy to compute
  - There are better sequences
- Quite difficult, depends on the sequence
  - And we do not know enough about it

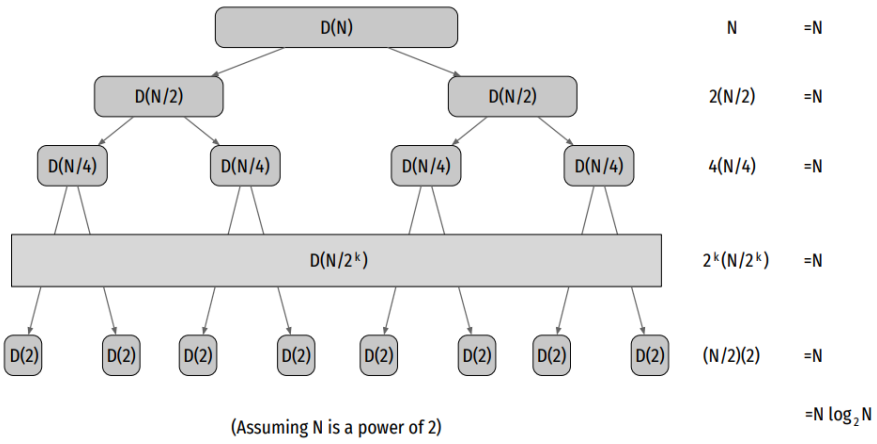
➢ Merge sort (Not in place and stable)

- Merge sort in 3 minutes
- Split the list in half
- Merge sort both halves recursively
- Merge the two sorted lists
- Divide and conquer
- We can merge two sorted lists in O(m + n), where m and n are the sizes of the two lists
- Advance pointers in the two lists independently
- Pick the smallest and add to the merged list



-
- Sorting

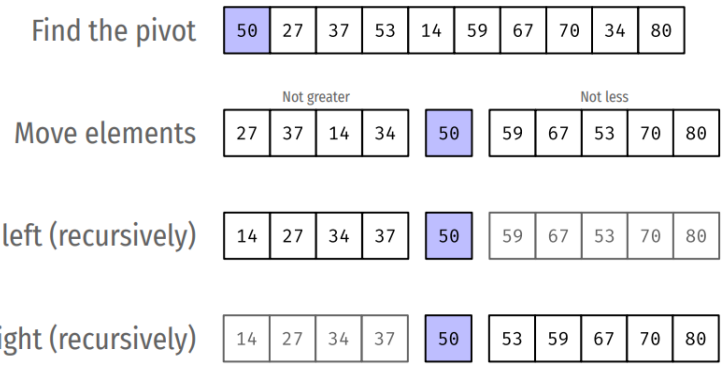
- When is a random list sorted
    - When it has 1 or 0 elements
- Divide lists until they have one element
- Then merge them together in sorted order
- 

- 

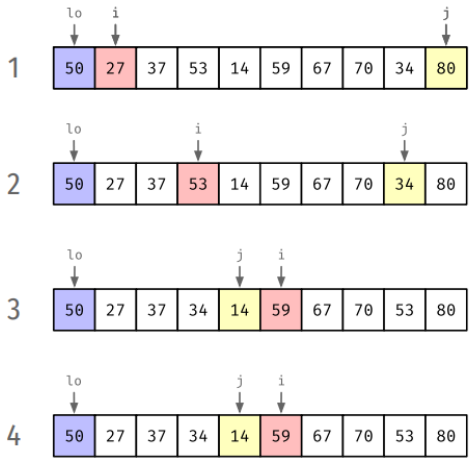
-

■ Analysis
- Almost perfect in terms of comparisons
- O(n log n)

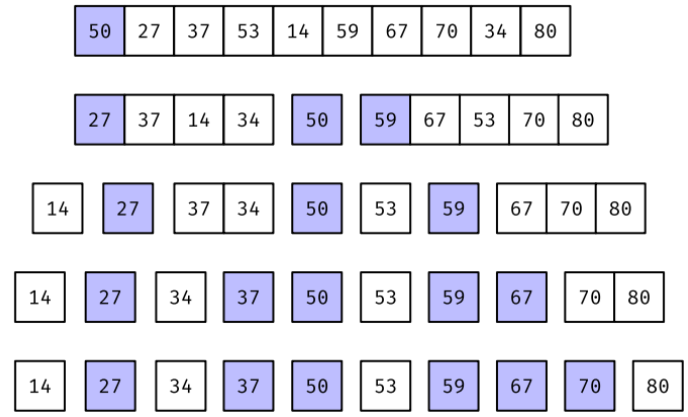
- Quick sort (In place and not stable)
  - Quick sort in 4 minutes
  - Divide and conquer, just like Mergesort
  - Split the input into two smaller parts
  - But split around a pivot value and ensure that
    - Values to the left are not greater than
    - And values to the right not less than the pivot
  - Avoids merge step



  -
  - Partition



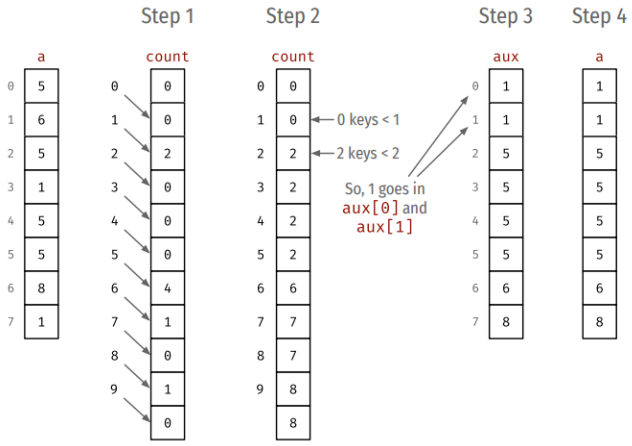
    -
  - Partition and sort



- Analysis
  - n log n average case
  - n^2 / 2 worst case
    - Worst case is rare
    - Ideally, we want the pivot to be the median
    - We can shuffle or approximate the median from lo, mid, hi

➢ Comparison-based sorts
- What is the lower bound of comparison-based sorting?
- Compare each value with every other value
  - Would suggest Ω(n^2)
  - We know that some algorithms are O(n log n)
- Ω(n log n)?
  - Would mean that merge and heap sort are (asymptotically) optimal
- The lower bound of O(n log n) for comparison-based sorting comes from decision trees. To correctly sort n elements, an algorithm must distinguish between n! possible permutations. Since a binary tree with depth d can represent at most 2^d leaves, it follows that d ≥ Ω(n log n). Hence, any comparison-based sorting algorithm requires at least Ω(n log n) comparisons to handle all permutations.

➢ Radix sort (Not in place and stable)
- Radix Sort | GeeksforGeeks
- We know that comparison-based sort is Ω(n log n)
- We can reduce this if we avoid comparing
- But how can we sort without comparing?
  - We can count



- Analysis
  - O(string length * number of strings)
  - Linear for short strings

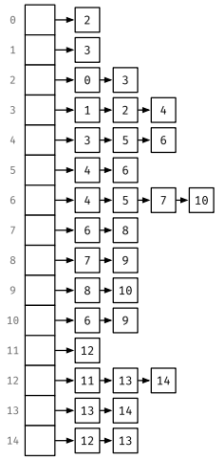
❖ Lecture 8 (Graphs) (Chapters: 8)

➢ Undirected

- Vertices are integers
- Edges can be:
  - List of edges, Adjacency matrix, or Adjacency list

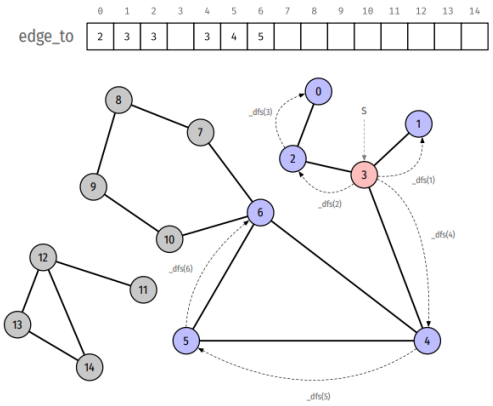
Representation	Space	Add edge	Has edge	Adj
List of edges	E	1	E	E
Adjacency matrix	V^2	1	1	V
Adjacency list	$E + V$	1	degree(v)	degree(v)

  - We often choose Adjacency list as real world graphs are often sparse
  - Adjacency List

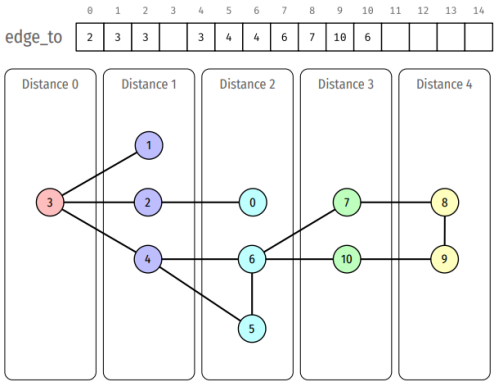


◆

- Depth First Search
  - The algorithm
    - Explores as far along as path as possible before backtracking
    - Visit a vertex and marks it as visited
    - Visits all unmarked vertices adjacent to it

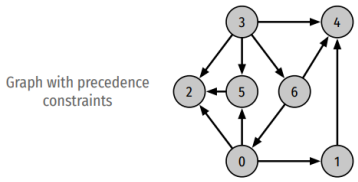


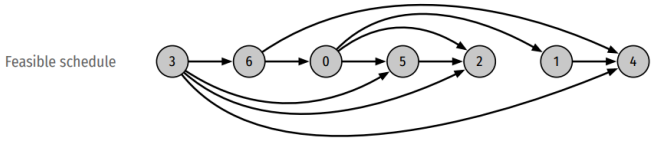
  - Breadth First Search
    - Visits vertices “distance by distance”



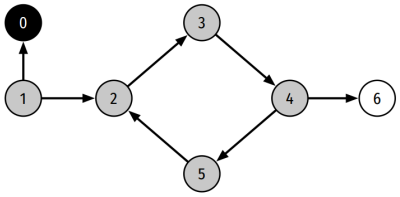
  - Connected Components
    - Two vertices, v and w are connected if there is a path between them
    - A connected component is a maximal set of connected vertices
    - Similar to Union-find, but we cannot use the same algorithms
  - Euclerian path is a cycle that uses every edge exactly once
    - This requires every vertex to have an even degree
    - Easy to solve O(E)
  - Hamiltonian path is a cycle that visits each vertex only once
    - Intractable (no efficient algorithm exists)
- Directed Graphs
  - Used for
    - One way streets, hyperlinks, legal move in a game, and control flow in programs
  - Every undirected graph is a directed graph with edges in both directions

- Topological sorting
  - Given a set of tasks and precedence constraints, in which order should the tasks be scheduled?
  - We can solve this with a directed graph where vertices are tasks, and edges give precedence constraints

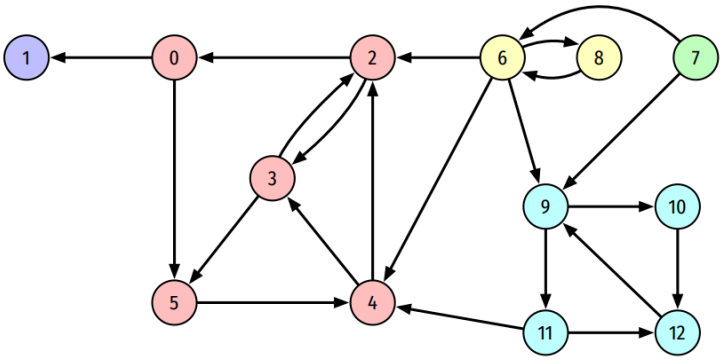




  - For this to work is must be directed acyclic graph
  - The algorithm
    - We start with a DFS to get a reverse postorder
    - Which we then reverse
    - Which is the topological order
- Finding cycles
  - If we have visited a node we mark it black, if we haven't it is white, and if we are processing it mark it as gray.
  - If during processing we run into a gray node we have a cycle.



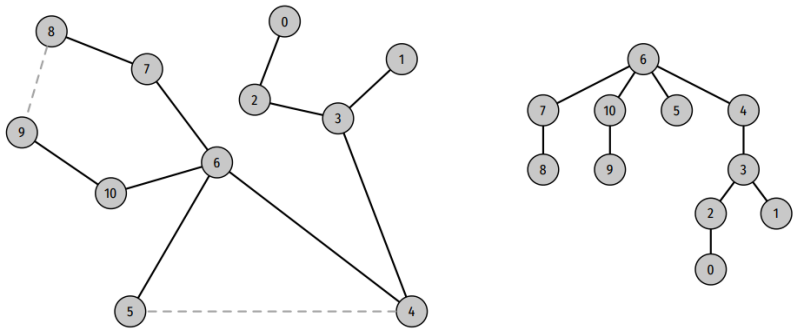
- Strongly-connected components
  - Two vertices, v and w are strongly connected if there are directed paths from v to w and w to v
  - A strong component is a maximal subset of strongly connected vertices



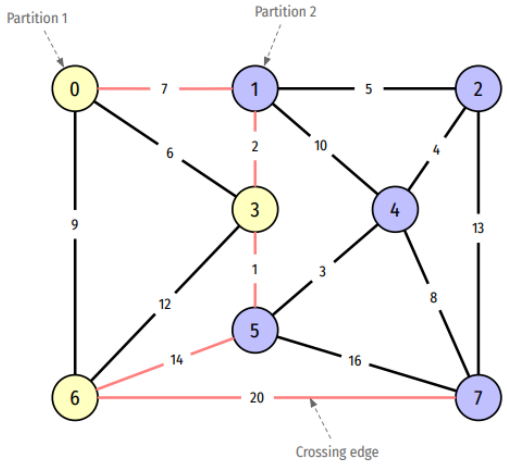
- Observation and Idea
  - The strong components are the same in the original graph and the reversed graph
  - So:
    - Compute the topological order of the reversed graph
    - Run DFS on the original graph on nodes in the order from step 1
  - This is the Kosarahu-Sharir algorithm which is used to find strongly connected components

➢ Spanning Trees

- A spanning tree is a subgraph of an undirected graph G which contains all the vertices of G
  - Tree, connected and acyclic
  - Spanning, includes all vertices



- There can exist multiple spanning trees
- Minimum spanning tree (MST)
  - Assume a connected, undirected graph with positive edge weights
  - A minimum spanning tree is the spanning tree with the lowest sum of edge weights
  - Given any cut, the crossing edge of minimum weight must be in the minimum spanning tree
  - So, to compute the MST, find a cut that does not have a crossing edge in the current tree
  - Add the minimum weight crossing edge to the tree
  - Repeat until V-1 edges are in the tree
  - Greedy algorithm
    - Pick the best choice at each step
    - Can work to find the global optimum
- A cut is away to partition the graph into two sets of vertices
- An edge that crosses the cut connects a vertex in one with a vertex in the other



- Kruskal's algorithm
  - Sort edges by weight (ascending order)
  - Add next edge to the tree if it does not create a cycle
  - How do we know if it creates a cycle or not?
    - We can use Union Find
    - It then uses the disjoint-set data structure to determine if these two vertices are already in the same set (i.e., part of the same connected component). If they are in the same set, adding the edge would create a cycle, and the algorithm skips this edge. If they are not in the same set, it means that adding this edge would connect two separate components without forming a cycle, so the edge is added to the minimum spanning tree.
- Prim's algorithm
  - Start with vertex 0
  - Grow the tree, T, greedily
    - Add the edge with minimum weight and exactly one endpoint in T
    - Repeat until V-1 edges
- Analysis
  - With binary heap, Prim is O(E log V) and Kruskal O(E log E)
  - So, Kruskal is better for sparse graphs, Prim is better for dense graphs

➢ Shortest Path

- Given a directed graph with edges weights, find the shortest path from s to t
- Above “source-sink”, can also be
  - Single source to every vertex
    - This is where you find the shortest path from a single source vertex 's' to every other vertex in the graph. Common algorithms for this include Dijkstra's algorithm

and Bellman-Ford algorithm. Dijkstra's algorithm works for graphs with non-negative edge weights, while Bellman-Ford can handle graphs with negative edge weights but detects negative weight cycles.

- Between all pairs of vertices
  - In this variation, you aim to find the shortest paths between all pairs of vertices in the graph. The Floyd-Warshall algorithm is a classic approach for this problem, and it can also handle graphs with negative edge weights, but not negative weight cycles.

- Cycles and negative weights can make it harder
- Single source
  - How can we compute?
    - Shortest path from s to every other vertex
    - Becomes a shortest path tree
    - We represent every edge in the tree using parent-link
      - Dist_to[v] is the length of the shortest path from s to v
      - Edge_to[v] is the last edge of the shortest path from s to v
    - We compute by relaxing edges
    - Assume we relax the edge v, w
      - Dist_to[v] is the shortest known path from s to v
      - Dist_to[w] is the shortest known path from s to w
      - Edge_to[w] is the last edge of the shortest known path from s to w
    - If v, w gives a shortest path to w through v
      - Update dist_to[w] and edge_to[w]
    - To compute the shortest path from a source vertex 's' to all other vertices in a directed graph, we use a process that creates a "shortest path tree." We maintain two arrays: 'Dist_to' stores the length of the shortest path from 's' to each vertex, and 'Edge_to' records the last edge on the shortest path to each vertex. We achieve this by repeatedly relaxing edges, which means updating 'Dist_to' and 'Edge_to' whenever we find a shorter path. This process continues until we've considered all vertices, resulting in a shortest path tree from 's' to every other vertex in the graph. Becomes a shortest path tree.
  - In what order should we relax edges?
  - Assume
    - No cycles
    - So, Edge-weighted Directed Acyclic Graphs (DAGs)
  - We can use topological order to compute shortest paths

- Cycles
  - The simple algorithm
    - ➢ No cycles, but positive and negative weights
  - What if we have cycles?
    - ➢ Dijkstra
  - But no negative weights
  - Bellman-Ford
    - ➢ What if we have no negative weights?
    - ➢ We can use Bellman-Ford but still no negative cycles
    - ➢ Relax all E edges V times
    - ➢ Notes: if dist_to[v] does not change in pass i, no need to relax any outgoing edges from v in pass i+1
    - ➢ We can use a queue
      - But we cannot have several copies of the same vertex in the queue at the same time!
- Dijkstra's algorithm
  - Consider vertices in increasing distance from the source
  - Add vertex to the shortest path tree and relax all outgoing edges from that vertex
  - Prim's

- Memoization is a technique used in computer science and programming to optimize algorithms, especially those involving recursive functions. It involves storing the results of expensive function calls and returning the cached result when the same inputs occur again. This can greatly improve the efficiency of algorithms that would otherwise be computationally expensive.
- Summary of spanning tree shit
  - Spanning Trees are fundamental concepts in graph theory. They are subgraphs of an undirected graph that contain all the original graph's vertices and form a tree structure, meaning they are connected and acyclic. There can be multiple spanning trees for a given graph, but one specific type of spanning tree is of great interest: the Minimum Spanning Tree (MST). The goal in finding an MST is to select a spanning tree with the lowest sum of edge weights.
  - To compute the MST, you can use greedy algorithms such as Kruskal's or Prim's. Kruskal's algorithm involves sorting the edges by weight and then adding edges one by one to the tree, making sure that they do not create a cycle. This is achieved using the Union Find data structure to check for the presence of cycles.
  - On the other hand, Prim's algorithm starts with an arbitrary vertex and grows the tree greedily by adding the edge with the minimum

weight that connects to the existing tree. It continues this process until you have V-1 edges in the tree, where V is the number of vertices.

- These algorithms have different time complexities and are suitable for different types of graphs. Kruskal's algorithm is better for sparse graphs, while Prim's algorithm is more efficient for dense graphs.
- Regarding Shortest Path problems, they involve finding the shortest path from a source vertex 's' to either a target vertex 't' (source-sink) or to every other vertex in the graph (single source to every vertex). For the latter case, Dijkstra's algorithm and Bellman-Ford algorithm are commonly used. Dijkstra's algorithm is suitable for graphs with non-negative edge weights, while Bellman-Ford can handle graphs with negative edge weights but detects negative weight cycles.
- If you need to find the shortest paths between all pairs of vertices, the Floyd-Warshall algorithm is a classic approach that can handle graphs with negative edge weights but not negative weight cycles.
- In the context of single source shortest path, you maintain two arrays: 'Dist_to,' which stores the length of the shortest path from 's' to each vertex, and 'Edge_to,' which records the last edge on the shortest path to each vertex. By repeatedly relaxing edges, you update 'Dist_to' and 'Edge_to' when a shorter path is found. This process continues until you've considered all vertices, resulting in a shortest path tree from 's' to every other vertex in the graph.
-
- When dealing with graphs that contain cycles, the approach depends on whether there are negative edge weights. For graphs with no cycles and negative weights, you can use a topological order to compute shortest paths. If cycles are present but no negative weights, Dijkstra's algorithm is appropriate. If negative weights are present but no negative cycles, the Bellman-Ford algorithm is suitable. And if there are no negative weights or negative cycles, you can use Bellman-Ford with fewer passes to improve efficiency.