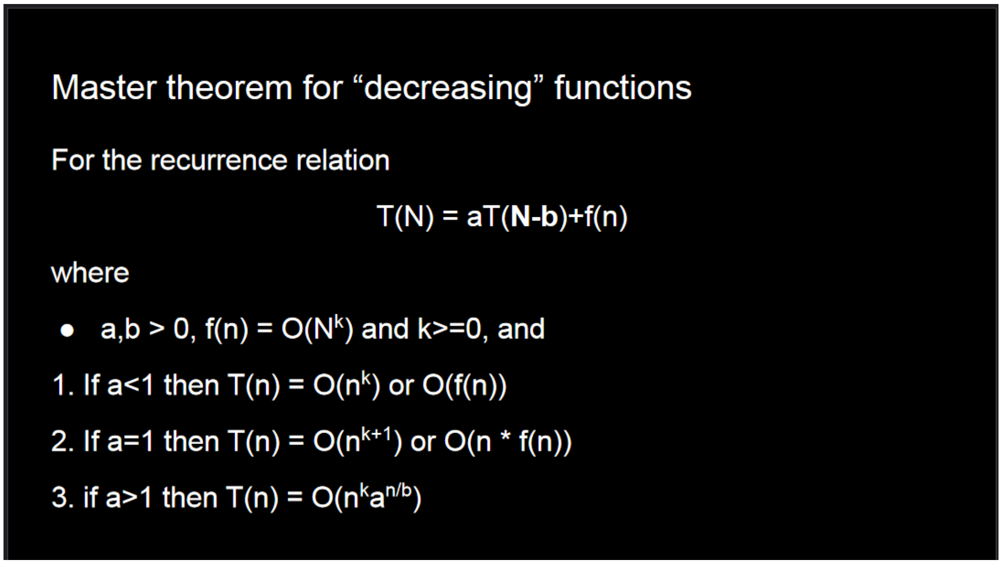
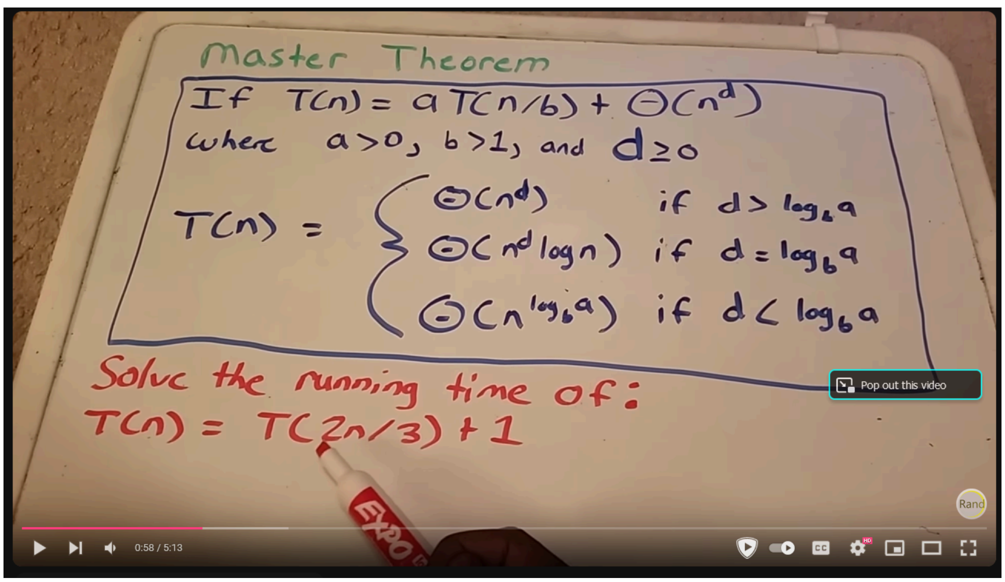
Master Theorem:





Functions to Recurrence Relation:

- **T(n)** = name of function such as **someFunc(n)**
- **C1, c2, etc.** = simple lines of code like print, comparison, etc.
- **T(n-1), T(n/2), T(n/4)** = recursive calls in code like **someFunc(n-1), someFun(n/2), someFunc(n/4)**
- **(n+1)** = for loops definition
- **N** = statements inside for loops since it's linear

Sorting Algorithms:

Average Time Complexities:

- Insertion Sort - O(n^2)

- Merge Sort - O(n log n)
- Quick Sort - O(n log n)
- Heap Sort - O(n log n)

Insertion Sort:

Pseudocode:

```
INSERTION-SORT(A, n)
for i = 1 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i − 1].
    j = i − 1
    while j ≥ 0 and A[j] > key
        A[j + 1] = A[j]
        j = j − 1
    A[j + 1] = key
```

Time/Space Complexity:

Time: **O(n^2)**

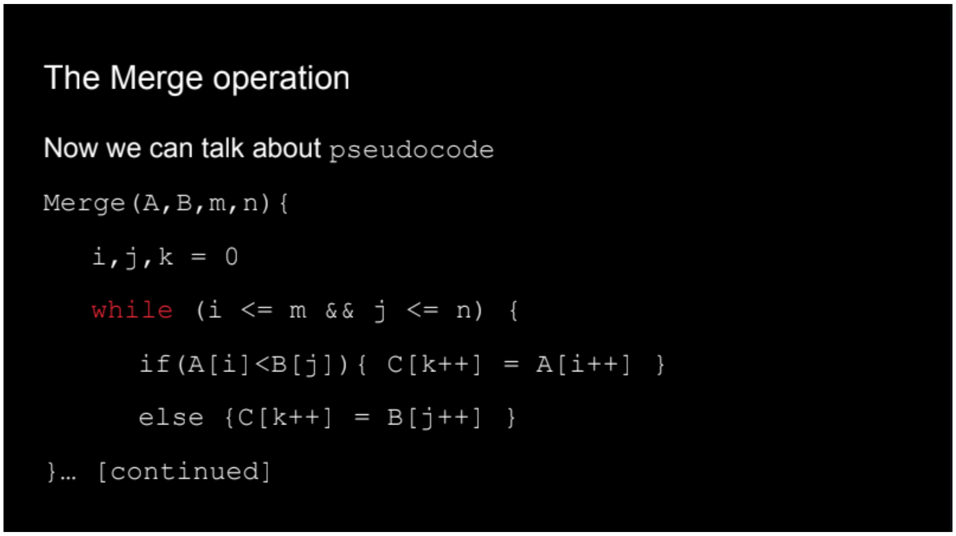
Space: **O(1)**

Mergesort:

Pseudocode:

Mergesort: The Algorithm

```
mergeSort(arr[], l, r)
    If l < r:
        mid = (l + r) / 2              Break array in half
        mergeSort(arr, l, mid)         mergeSort left half
        mergeSort(arr, mid + 1, r)     mergeSort right half
        merge(arr, l, mid, r)          Merge results
```



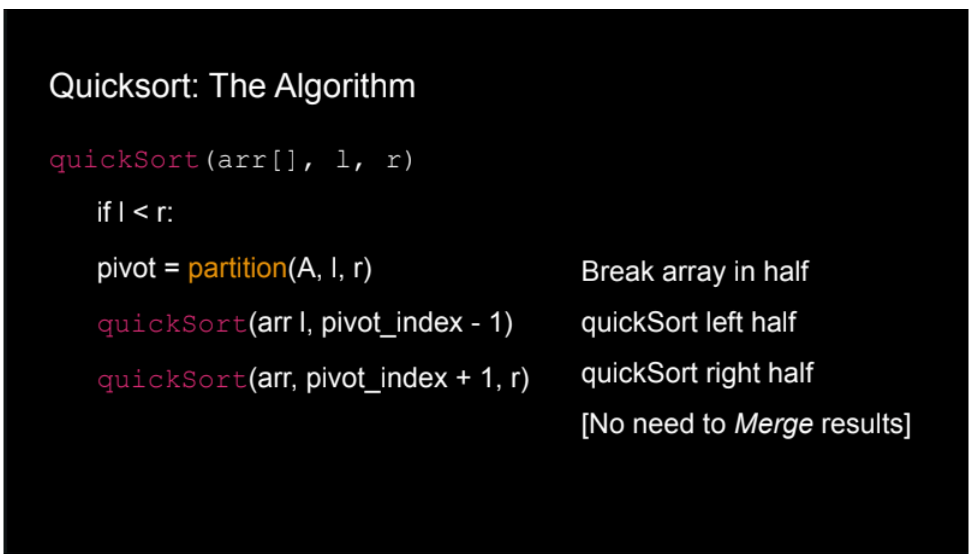
Time/Space Complexity:

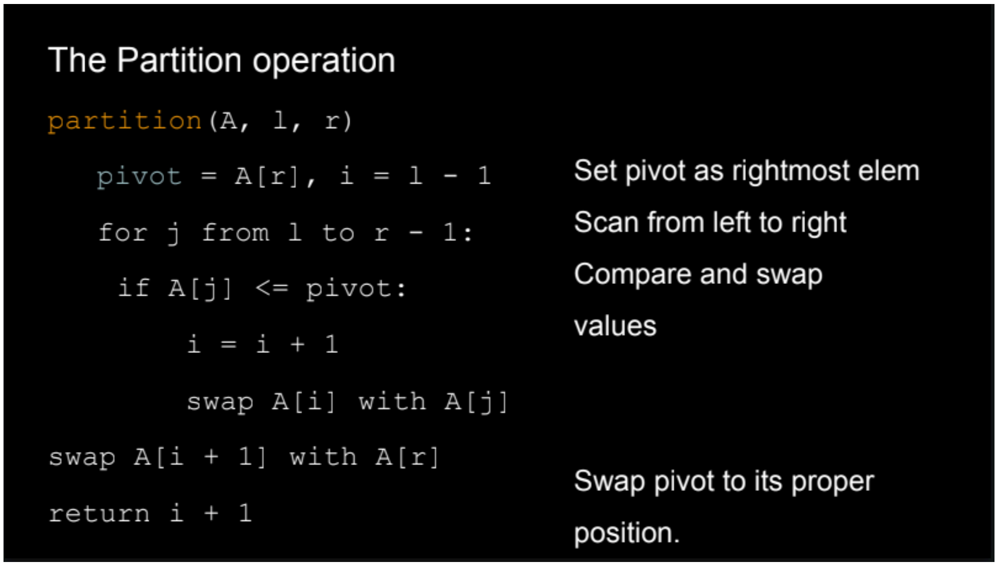
Time: **Θ(n log n)** for best, average, worst

Space: **Θ(N)** which is m+n elements (can be thought of as m=n=N)

Quicksort:

Pseudocode:





Time/Space Complexity:

Time: **Θ(n log n)** - best, **Θ(n log n)** - average, **Θ(n^2)** - worst

Space: **Θ(n)** - worst

Things to know:

- Quicksort is UNSTABLE

Heap sort:

Time/Space Complexity:

Time: **Θ(n log n)** -> Θ(n) for buildHeap, Θ(log n) for delete

Space: **Θ(1)** -> in-place algorithm

Things to know:

- It's an in-place algorithm (doesn't need extra space like mergesort)
- It's UNSTABLE (equivalent elements may be swapped)
- Slower than quicksort in practice

Counting Sort:

Time/Space Complexity:

Time: **Θ(n + k)**

Space: **Θ(n + k)**

Radix Sort:

Time/Space Complexity:

Time: **Θ(n*d)**

Space: **Θ(n + k)**

Bucket Sort:

Time/Space Complexity:

Time: **Θ(n^2)**

Space: **Θ(n + k)**

Hashmaps:

Hash Collisions:

- Linear probing
  - Traverse linearly, put element in next empty space
  - Let's say C also hashes to index 2
  - It will put element into next empty space
- Quadratic Probing
  - Quadratic Probing is similar, but we increment faster if we keep running into collisions.
  - If the slot hash(x) is full, then we try hash(x) + 1*1
  - If hash(x) + 1*1 is also full, then we try hash(x) + 2*2
  - If hash(x) + 2*2 is also full, then we try hash(x) + 3*3
  - Rinse and repeat until you find an empty spot
- Separate Chaining
  - Uses additional data structures to allow for multiple values at the same index. E.g. linked lists
- Double Hashing
  - Uses a second hash function to reduce the chance of a collision happening

Time/Space Complexity:

Time: **Θ(1)** for inserting, deleting, and referencing

Space: **Θ(N)**

Trees:

Binary Search Trees:

Time/Space Complexity:

Operation	**Best**	**Average**	**Worst**
Search	Θ(1)	Θ(log n)	Θ(n)
Insert	Θ(1)	Θ(log n)	Θ(n)
Delete	Θ(1)	Θ(log n)	Θ(n)

Worst case happen in skewed BSTs

Red-Black Trees:

Properties:

- Every node is either red or black
- Root is black
- NIL nodes are black
- Red node doesn’t have red child
- Every path from given node to any of its descendant NIL nodes goes through same number of black nodes

Applications:

- Completely fair scheduler - process scheduler for linux kernel
- Handling hash collisions in Java (separate chaining)
- Speeding up other algorithms

Time/Space Complexity:

Time: **Θ(log n)** worst case for all operations

Graph Algorithms:

Graph Representation:

Adjacency Matrices:

- Use more memory **Θ(n^2)**
- Fast lookup and checks for presence of edges **O(1)**
- Slow to iterate over all edges
- Slow to add/delete a node **O(n^2)**
- Fast to add a new edge **O(1)**

Adjacency List:

- Memory usage depends more on number of edges, not nodes
- Helps when graph is **sparse**
- Slow lookup and checks for presence of edges **O(k)** (k = num of neighbor nodes)
- Faster to iterate all over edges
- Also fast to add/delete a node
- Fast to add new edge **O(1)**

When to use what:

- Use adjacency lists when expecting a sparse graph, or when you need fast lookup for neighbors of a vertex. Most real life situations will generate sparse graphs
- Use adjacency matrices when expecting a dense graph, or when you need fast lookup for edges
- Rule of thumb: If density (edges/nodes^2) goes over 1/64 (for 32-bit computers), use adjacency matrix

BFS/DFS:

Data structures:

- BFS uses queue
- DFS uses stack

Pseudocode (BFS):

- Create queue
- Mark vertex as visited and put into queue
- While queue is not empty
  - Remove u (front) of queue
  - Mark and enqueue neighbors of u

Pseudocode (DFS):

- Create stack
- Push vertex v into stack
- Mark vertex v as visited
- While stack is not empty
  - Pop vertex v stack
  - For all neighbors of v
    - If neighbor u is not visited
      - Mark u as visited
      - Push u into stack

Time/Space Complexity:

Time: **O(V+E)** for best, average, worst case for both

Space: **O(V)** for both

Topological Sort (Khan's Algorithm):

Pseudocode:

- Add all nodes with in-degree 0 to queue
- While queue is not empty:
  - Remove node from queue
  - For each outgoing edge from node, decrement in-degree of the destination node by 1
  - If the in-degree of a destination node becomes 0, add it to the queue

Other things to know:

- Only works on DAG
- Topological sort can be implemented using DFS and BFS
- Intuition for both:
  - Traverse graph to find two types of nodes:
    - Nodes with no outgoing edges go last

            - Nodes with no incoming edges go first

Time/Space Complexity:

Time: **O(V+E)**

Space: **O(V)** for queue/stack

MST Algorithms:

Prim's Algorithm:

How to run:
- Add starting vertex v to visited list
- Examine all neighbors of v
- Pick cheapest cost that connects to an unvisited node
- Add that node to visited list
- From visited node(s), repeat step 3-5

Pseudocode:
- Pick starting vertex
- Create queue
- Create visited boolean array
- While queue is not empty
    - Explore neighboring vertices
    - Pick cheapest cost and mark as visited
    - Repeat as long as it doesn't create a cycle

Time/Space Complexity:

Time: **O(V^2)** for adj. matrix, **O(V log V + E log V)** for adj. list

Kruskal's Algorithm:

How to run:
- Pick smallest edge
- Repeatedly look for smallest edge that doesn't create a cycle

Pseudocode:
- Create empty set to store mst
- Create priority queue and add starting vertex with key 0
- While PQ is not empty:
    - Extract vertex u with minimum key from PQ.
    - Add u to set.
    - For each neighbor v of u:
        - If v is not in S and weight(u, v) < key(v):
            - Update key(v) with weight(u, v).
            - Update parent(v) with u.
- Return set.

Time/Space Complexity:
Time: **O(E log E)**

Other notes:

Data Structures:

- Prim's uses **lists and heaps**
- Kruskal's uses **disjoint sets**
    - A list of disjoint sets
    - Uses union-find

Applications:

- Constructing trees for broadcasting in computer networks
    - On ethernet networks: spanning tree protocol
- Curvilinear feature extraction in computer vision
- Cluster analysis:
    - Clustering points in the plane (K means)
    - Graph-theoretic clustering (social networks)
    - Clustering gene expression data

Shortest Path Algorithms:

Intuition:

- 3 algorithms to find shortest path and what they work on
    - **BFS** - designed for unweighted graphs, can be modified (use heap instead of queue)
    - **DFS** - may take longer to complete in case of cycles
    - **Topological Sort** - only works on DAGs

Dijkstra's Algorithm:

Pseudocode:

- For each vertex in graph
    - Set distance to infinity
    - Set previous to undefined
- Set distance of starting vertex to 0
- While queue is not empty
    - Remove vertex u with smallest distance
    - For each neighbor v of u
        - Relax edges

Time/Space Complexity:
Time: **Θ(V^2)** for array implementation (Improves to **Θ(E + V log V)** if we use fibonacci heap which is fastest sssp algorithm known today)
Space: **O(V)** in both cases

Other things to know:

- Doing updates is unnecessary work. We can make more intelligent choices by comparing edges in terms of (edge weight + shortest distance to get to that edge) instead of just edge weight.
- Dijkstra's will fail if there is a negative weight! Fix for this is actually a simple modification!

Bellman-Ford:

Pseudocode:

```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

- Initialize graph like dijkstra
- For i = 1 to v-1
    - Relax edges
- Check for negative cycles

Time/Space Complexity:

Time: **O(E)** - Best, **O(V*E)** - average, worst

Space: **O(V)**

What it is:

- Essentially Dijkstra's algorithm but is fixed for negative weights
- Basic idea is that the longest path in a graph is V-1 length
- If you are looking at a path length of V or more, during iteration, some cycle is present in your graph
- All we need to do is systematically look through our graph V-q times and update the shortest path that many times.

A*:

What it is:

- The general case of Dijkstra's algorithm
- Basic idea is instead of just looking at edge weights, we add a heuristic function value to each edge weight
- Algorithm is essentially:
    - Calculate/approximate the heuristic

- Do Dijkstra's with heuristics applied to weights

Heuristic:

- Heuristic is like hash functions, it could be anything
- Examples:
    - Euclidean distance
    - Manhattan distance
    - Arbitrary coin flip
    - Traffic on a road
- They are expensive to compute/store for large graphs, so we approximate for A*
- Quality of heuristic defines how quickly you descend on the right results

Applications:

- Dijkstra and A* is what Google Maps uses
- They are used in:
    - All of graph-based ML
    - Quant trading
    - Social network analysis
    - Pathfinding in video games
    - City planning
    - And many more!

Floyd-Warshall Algorithm:

Pseudocode:

Floyd-Warshall Algorithm Pseudocode

dist is a |V| × |V| array of minimum distances initialized to ∞ or nil

```
for each edge (u, v) do
    dist[u][v] ← w(u, v)  // The weight of the edge (u, v)
for each vertex v do
    dist[v][v] ← 0
for k from 1 to |V|
    for i from 1 to |V|
        for j from 1 to |V|
            if dist[i][j] > dist[i][k] + dist[k][j]
                dist[i][j] ← dist[i][k] + dist[k][j]
```

The "heart" of the algorithm

- N = num of vertices
- A = matrix
- For k = 1 to n
    - For i = 1 to n
        - For j = 1 to n
            - Distance[i][j] = min(distance[i][j], distance[i][k] + distance[k][j])

Time/Space Complexity:

Time: **Θ(n^3)**

Space: **O(V^2)**

Other notes:

- Algorithm is best suited for dense graphs
- Exhaustive search

Johnson's Algorithm:

What it is:

- Makes use of Dijkstra's and Bellman-Ford
  - Run Bellman-Ford on graph and do a transformation to remove negative edges
  - Run Dijkstra's for every vertex on the resultant graph
- Faster than Floyd-Warshall for sparse graphs

Dynamic Programming:

Intuition:

- Fibonacci normally implemented using recursion time complexity is **O(2^n)** and space complexity is **O(1)**
- We can improve time complexity by using space to store previously known results which is a tradeoff (but is worth it)
- Now it would be **O(n)** time and **O(n)** space complexity
- Not an algorithm, but a paradigm (way of thinking)
- Basic idea is that the problem can be divided into subproblems but unlike divide and conquer, the problems overlap
- Overlapping subproblems is the first property that hints that we can use dynamic programming to solve a particular problem.
- When in doubt, always think about whether we are repeating unnecessary work, to determine whether we can use DP for a problem.

Memoization:

- Memoization makes subsequent runs of the function/algorithm much, much faster.
  - If we know fibo(4), then fibo(4) or fibo(3) becomes an O(1) lookup since we already computed it in a previous run
- Another cool thing about memoization is that it's guaranteed to use just enough memory to get to the required result
  - We didn't calculate fibo(6) because we didn't need to, for fibo(4)

Tabulation:

- More work to do since we may end up doing more work than needed to arrive at the optimal solution
- But the hope is that if we solve each subproblem optimally, we arrive at an optimal solution
- Optimal substructure is actually the 2nd property we need to do problems in DP (also helps with greedy algorithms)
- Ideally, we should always use memoization but sometimes tabulation can't be avoided.

P and NP:

Every problem (sorting, searching, tree traversal, topological sort, MST, etc.) we've looked at so far has been solvable algorithmically, in polynomial time. This is formally called **P** or **P-class** problems/algorithms

Sudoku 3x3 is easy to bruteforce/solve, 9x9 takes longer, and so on.
Verifying the correctness of these is always fast. These problems are called NP (nondeterministic-polynomial)

If P = NP, then every problem can pretty much be solved
If P != NP, then we will know for sure that some problems are hard to solve

"Algorithmica" - P = NP, every problem has a nice algorithm
"Heuristica" - P != NP, but most hard problems are tractable
"Cryptomania" - P != NP, most problems are too hard