Hash Tables: O(1) insertion and search (and deletion in some cases) for big space of keys. Keys are numeric representations of relevant portions of the data.
Why not arrays? Array must be as large as universe of keys, but universe of keys is often much larger than number of actual keys. E.g. if we're indexing surnames <10 characters, space of keys is 26^{10} when census only lists ≈89k possible keys.
Load of a hashtable: m = number of possible entries in table, n = number of keys stored in table, a = n/m is hashtable's load factor. Small a = table is wasteful.
Hash Function: A hash function is a function that maps the universe of keys to the slots in the hashtable. If U is the universe of keys and m is an index in the hashtable, then a hash function h: U → m. m << $|U|$.
Collisions: If m < $|U|$, two keys can map to the same position in a hashtable (collisions are inevitable, pidgeonhole principle). Collision occur when h(x) = h(y), but x ≠ y. Hash function should minimize number of collisions.
Collision Resolution by Chaining: Hashtable is an array of linked lists. When collision occurs, element is added to linked list. If two entries x ≠ y have the same hash value h(x) = h(y), then T(h(x)) will contain a linked list with both values. Operations: ChainedHashInsert(T, x) → insert x at head of linked list T[h(x)] (Θ(1)), ChainedHashDelete(T, x) → delete x from list T[h(key[x])] (O(len of chain)), ChainedHashSearch(T, x) → search for x in list T[h(x)] (O(len of chain)).
Length of chain: Average case depends on how well hash function distributes keys. Simple uniform hashing = elements are equally likely to be in any slot. With this property, average chain length = n keys / m slots = a.
Search average running time: If key isn't in table, must search all entries: $\Theta(1+a)$. If key is in table, search on average half the entries: $O(1+a)$.
What makes hash function good? Simple uniform hashing property. Deterministic: h(x) always returns same value. Low cost: if calculating is expensive (e.g. log n), no benefit from table.
Division method: Hash function where h(k) = k mod m. Don't use power of two as m. If m = 2^p, it will just be lower p bits of value. Good rule of thumb is to make m prime number that isn't too close to power of 2. Pros: quick to calculate, easy to understand. Cons: keys close to each other will end up close in hash table.
Multiplication method: Multiply key by a constant 0 < A < 1 and extract fractional part of kA, then scale by m to get index: h(k) = $\lfloor m(kA - \lfloor kA \rfloor) \rfloor$. Common choice is m = power of two and A = $(\sqrt{5}-1)/2$.
Open addressing: Array of linked lists can be inefficient/hassle. Instead, keep hash table as array. Handle collisions by computing another slot to examine.
Hash functions with open addressing: Hash function must define a probe sequence (list of slots to examine when searching/inserting). Hash function takes an additional parameter i (number of collisions that have already occurred). Hash function must be a permutation of every hash table entry.

```
Hash-Insert(T, k)
    i, j <- 0, h(k, i)
    while i < m - 1 and T[j] != null:
        i <- i + 1
        j <- h(k, i)
    if T[j] == null:
        return j
    else
        error "hash is full"
```

```
Hash-Search(T, k)
    i, j <- 0, h(k, i)
    while i < m - 1 and
          T[j] != null and T[j] != k:
        i <- i + 1
        j <- h(k, i)
    if T[j] == k:
        return j
    else:
        return null
```

Open addressing (delete): Mark nodes as "deleted" (instead of null) and modify search procedure to keep looking if "deleted" node is seen and insert procedure to fill in "deleted" entries. Since search times increase, use chaining if deleting a lot.
Probing schemes: Linear probing — Go to next slot upon collision. h(k, i) = (h(k) + i) mod m. Runs of occupied slots build up & tend to grow (primary clustering). Quadratic probing — Probe based on quadratic function. h(k, i) = (h(k) + $c_1 i$ + $c_2 i^2$) mod m. Must pick constants & m such probe sequence is valid. Potential secondary clustering. Double hashing — probe sequence determined by a second hash function. h(k,i) = (h_1(k) + i(h_2(k))) mod m.
Runtime of insert/search for open addressing: Depends on function/probe sequence. Worst case = O(n), visit every full entry before finding opening. On average, compute E[probes] = $1 + a + a^2 + a^3 + ... = \sum_{i=0}^{m} a^i < \sum_{i=0}^{\infty} a^i = \frac{1}{1-a}$.
Hashtable size: Generally about 1/2 full. When table gets full, either create a new one (one expensive insert) or create a new hashtable in an amortized way (no single operation will be very expensive).
BSTs: a parent's value is greater than all values in left subtree and less than or equal to all the values in right subtree and children are also binary search trees. Min is leftmost value and max is rightmost value.
BST Operations: Search(T, k), Insert(T, k), Delete(T, x), Minimum(T), Maximum(T), Successor(T, x), Predecessor(T, x), Median (T)

```
BSTSearch(x, k)
    if x == null or k == x
        return x
    elseif k < x
        return BSTSearch(Left(x), k)
    else
        return BSTSearch(Right(x), k)
```

BSTSearch Running Time - worst case = Θ(height of the tree), average case = O(height of the tree), best case = O(1). BSTInsert running time — O(height of the tree). For height of tree, the worst case is a twig and best case is a complete tree where height is log n.

```
BSTInsert(T, x)
    if Root(T) == null
        Root(T) <- x
    else
        y <- Root(T)
        while y != null
            prev <- y
            if x < y
                y <- Left(y)
            else
                y <- Right(y)
        Parent(x) <- prev
        if x < prev
            Left(prev) <- x
        else
            Right(prev) <- x
```

```
InorderTreeWalk(x)
    if x != null
        InorderTreeWalk(Left(x))
        print x
        InorderTreeWalk(Right(x))
```

```
PreorderTreeWalk(x)
    if x != null
        print x
        PreorderTreeWalk(Left(x))
        PreorderTreeWalk(Right(x))
```

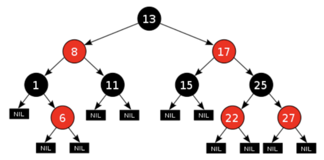
Successor and predecessor:

- Predecessor = max node of those smaller than node, so rightmost elem of left subtree. If no left subtree, the predecessor is first parent smaller than the node.
- Successor = min node of those larger than node, so leftmost elem of right subtree. If no right subtree, the successor is first parent bigger than the node. Runtime = O(height of tree).

```
Successor(x)
    if Right(x) != null
        return BSTMin(Right(x))
    else
        y <- Parent(x)
        while y != null and
        x == Right(y)
            x <- y
            y <- Parent(y)
    return y
```

Node deletion BST (three cases): no children → delete node, one child → splice out node, two children → replace x with its successor because larger than left subtree but less than or equal to right subtree

Red-black trees: Most operations run in O(height of tree). To ensure trees stay balanced, use AVL, 2-3-4 or red-black trees. Red black trees: (1) Every node is either red or black (2) Root is black (3) Leaves (NIL) are black (4) If a node is red, both children are black (5) For every node, all paths from the node to descendant leaves contain same number of black nodes.



Red-black trees vocab:

- h(x): height of node x: number of edges in longest path from x to a leaf
- bh(x): black height of node x: number of black nodes on a path from x to leaf (not including x) — h(root) = 4, bh(root) = 2

Bounding the height of red-black trees:

- Claim 1: for every node x, bh(x) ≥ h(x) / 2. Proof: in the worst case, nodes alternate red and black with root and leaf being black so minimum number of black nodes would be $\frac{h(x)}{2}+1$. Thus we can say that the statement holds since the black height does not include x.
- Claim 2: Claim 2: The subtree rooted at any node x contains at least $2^{bh(x)}-1$ internal (non-leaf) nodes.
  - Base Case (leaf: h(x) = 0): bh(x) = 0. $2^0 - 1 = 1$.
  - Inductive Case (h(x) > 0): IH = at least $2^{bh(y)}-1$ internal nodes ∀ y that are subtrees of x. If x is red: bh(child(x)) = bh(x) - 1. If x is black: bh(child(x)) = bh(x) or bh(x) - 1. Thus, bh(child(x)) ≥ bh(x) - 1. We can use this fact and our IH to show that there are $(2^{bh(x)-1}-1) + (2^{bh(x)-1}-1) + 1 = 2^{bh(x)}$ - 1 internal nodes.
- $n \geq 2^{bh(x)} - 1 \rightarrow n \geq 2^{h(x)/2} - 1 \rightarrow n+1 \geq 2^{h(x)/2} \rightarrow h(x) \leq 2\log(n+1)$. Thus if we maintain the red-black properties the height of tree is $O(\log n)$.

Extensible array: Want to support adding to array. To do efficiently, can allocate some extra memory and when full, double size and copy. Runtime of copy adds is $\Theta(n)$, O(1) otherwise. Want to figure out avg runtime of add in worst case:

Insertion	1	2	3	4	5	6	7	8	9	10
Size	1	2	4	4	8	8	8	8	16	16
Basic Cost	1	1	1	1	1	1	1	1	1	1
Double Cost	0	1	2	0	4	0	0	0	8	0

Amortized analysis: In general: `total_cost(n) = basic_cost(n) + double_cost(n)`. Here, `basic_cost(n)` = n and `double_cost(n)` = 1 + 2 + 4 + ... + n = 2n. Thus, `total_cost(n)` = 3n. Over n operations: amortized O(1).
Amortized analysis vs. worst case: Add's worst case still O(n)! Amortized analysis gives cost of n operations (i.e. avg cost) not cost of individual operations.
Extensible arrays: what if instead of doubling the array, we instead increase the array by a fixed amount (call it k) each time. The amortized runtime is no longer O(1). While the `basic_cost(n)` = O(n), `double_cost(n)` = k + 2k + 3k + 4k + 5k + . . . + n = $\sum_{i=1}^{n/k} ki = k\sum_{i=1}^{n/k} i = k\frac{(n/k)(n/k+1)}{2} = O(n^2)$. Thus `total_cost(n)` = O(n) + $O(n^2) = O(n^2)$. Amortized = O(n).
Accounting method: Each operation has an amount we charge. Deduct from that charge actual cost of operation. If anything left over, put in the bank. Operations may use bank to offset cost. Key idea: charge more for low-cost operations and save to offset costly operations. E.g. add to list, with insert charging 3 = O(1):

Insertion	1	2	3	4	5	6	7	8	9	10
Size	1	2	4	4	8	8	8	8	16	16
Cost	1	2	3	1	5	1	1	1	9	1
Bank	2	3	3	5	3	5	7	9	3	5

Binary heap: Max heap is binary tree where value of parent ≥ value of children. Additional restriction is: all levels of tree are complete except last.
Operations: Maximum(S) — return largest elem in set, ExtractMax(S) — return & remove largest elem in set, Insert(S, val) — insert val into set, IncreaseElement(S, x, val) — increase value of elem x to val, BuildHeap(A) — build heap from array
Heap representation: We can represent a heap as an array. `Parent(i) = return` ⌊ i/2 ⌋, `Left(i) = return 2i`, `Right(i) = return 2i + 1`.
Graphs: a graph is a set of vertices V and a set of edges (u, v) ∈ E where u, v ∈ V. Undirected — edges don't have direction. Directed — edges have a direction. Weighted — edges have associated weights. Path — a list of vertices $p_1, p_2, ..., p_k$ where there exists an edge $(p_i, p_{i+1}$ ∈ E. A simple path has no repeated vertices.

Cycle — a subset of the edges that form a path s.t.c first and last node are the same. Connected — every pair of vertices is connected by a path. Strongly connected (directed graphs) — every two vertices are reachable by a path.

Different types of graphs: Tree = connected, undirected graph without any cycles. DAG = directed, acyclic graph. Complete graph = an edge exists between every node. Bipartite graph = a graph where every vertex can be partitioned into two sets X and Y such that all edges connect a vertex u $\in$ X and a vertex v $\in$ Y.

Representing graphs:

- Adjacency list: Each vertex u $\in$ V contains an adjacency list of the set of vertices v such that there exists an edge (u,v) $\in$ E. Use for sparse graphs. Space efficient. Must traverse adjacency list to discover if edge exists.
- Adjacency matrix: a $|V|$ x $|V|$ matrix A such that A_{ij} is 1 if (i, j) $\in$ E and 0 otherwise. Dense graphs. Constant time lookup to discover if edge exists. Simple to implement. For unweighted graphs, only requires boolean matrix.
- Sparse adjacency matrix: rather than using adjacency list, use adjacency hashtable. Constant time lookup. Space efficient. Bad for dense graphs.
- If the graphs are weighted: for adjacency lists, store weight as an additional field in the list. For matrix, store weight instead of 1.

BFS:

```
TreeBFS(T)
    Enqueue(Q, Root(T))
    while !Empty(Q)
        v <- Dequeue(Q)
        Visit(v)
        for all c in Children(v)
            Enqueue(Q, c)
```

Runtime is $\Theta(|V| + |E|)$ for adjacency lists and $\Theta(|V|^2)$ for adjacency matrices if connected.

```
BFS(G, s)
    for each v in V
        dist[v] = infinity
    dist[s] = 0
    Enqueue(Q, s)
    while !Empty(Q)
        u <- Dequeue(Q)
        visit(u)
        for each edge (u, v) in E
            if dist[v] == infinity
                Enqueue(Q, v)
                dist[v] <- dist[u] + 1
```

DFS:

```
TreeDFS(T)
    Push(S, Root(T))
    while !Empty(S)
        v <- Pop(S)
        Visit(v)
        for all c in Children(v)
            Push(S, c)
```

Runtime of DFS is $\Theta(|V| + |E|)$.

```
DFS(G)
    for all v in V
        visited[v] <- false
    for all v in V
        if !visited[v]
            DFS-Visit[v]

DFS-Visit(u)
    visited[u] <- true
    for all edges (u, v) in E
        if !visited[v]
            DFS-Visit(v)
```

Dynamic Programming: Problem solvimg method where optimal solutions can be defined in terms of optimal solutions to subproblems & subproblems overlap.

1a. Optimal Substructure: show optimal solutions to problem incorporate optimal solutions to related subproblems.
1b. Recursive definition: use to recursively define value of an optimal solution
2. DP solution: describe dp table, size, initial vals, fill order, where solution is
3. Analysis: analyze space requirements, running time

Fibonacci: Naive implementation: two recursive calls for each call. This creates a full binary tree of depth n, so runtime is $O(2^n)$. We have optimal substructure and recursive definition trivially. Table is all of the problems F(1)...F(n-1). Fill in table from F(1) to F(n). space = $\Theta(n)$, time = $\Theta(n)$. DP solution:

```
Fibonacci-DP(n)
    fib[1], fib[2] <- 1, 1
    for i <- 3 to n
        fib[i] <- fib[i - 1]
                  + fib[i - 2]
    return fib[n]
```

BST Counting: Count unique BSTs we can create using numbers 1 to n?

1a. By definition since binary trees are recursive structures.
1b. T(n) = $\sum_{i=1}^{n} T(i-1) * T(n-i)$
2. Smallest possible subproblems: T(0) = 1, T(1) = 1. Table is T(0)...T(n-1). Fill in small to big.
3. space = $\Theta(n)$, time = $\Theta(n^2)$

```
BST-Count-DP(n)
    c[0], c[1] <- 1, 1
    for k <- 2 to n
        c[k] <- 0
        for i <- 1 to k
            c[k] <- c[k] +
                    c[i - 1] * c[k - i]
    return c[n]
```

Longest Common Subsequence: For a sequence X = $x_1, x_2, ..., x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, ..., i_k)$ where $1 \leq i_1 < i_2 < ... < i_k \leq n$. Given two sequences x and y, what is the lcs?

1a. Proof by contradiction. Assume $s_1, s_2, ..., s_m$ is the LCS(x, y), but $s_2, ..., s_m$ isn't the optimal solution to LCS(substring_after(s_1, X), substring_after(s_2, Y)). If this was true, we could make a longer subsequence by s_1 LCS(substring_after(s_1, X), substring_after(s_2, Y)). Contradiction!
1b. LCS(x, y) = 1 + LCS($x_{1...n-1}$, $y_{1...m-1}$) if $x_n = x_m$ and max(LCS($x_{1...n-1}$, y), LCS(x, $y_{1...m-1}$)) otherwise.
2. 2d table. If characters are equal, do 1 + diagonal up/left. Otherwise, take max of going left or up. Final answer bottom rightmost element.
3. space = $\Theta(nm)$, time = $\Theta(nm)$

Rod Splitting: Input: a length n and a table of prices for i = 1,2,...m. Output: maximum revenue obtainable by cutting up the rod and selling the pieces.

Length	1	2	3	4	5	6	7	8	9	10
Price	1	5	8	9	10	17	17	20	24	30

1a. Proof by contraction. $\{l_1, l_2, ..., l_m\}$ is a solution to n but $\{l_2, ..., l_m\}$ is not a solution to n - l_1. If that was true, some solution to n - l_1 exists where the sum of the prices of the lengths is greater than that for $\{l_2, ..., l_m\}$. We could add l_1 to this solution and get a better solution to the n problem. Contradiction!
1b. R(n) = $max_{i:n-l_i \geq 0}$ $\{p_i + R(n - l_i)\}$
2. 1d table. Need to solve subproblems 0...n-1. Smallest subproblem is R(0) = 0. Fill in small to large.
3. space = $\Theta(n)$, time = $\Theta(nm)$

```
DP-Rod-Splitting(n)
    r[0] <- 0
    for j <- 1 to n
        max <- 0
        for i <- 1 to m
            if l_i <= j
                p <- p_i + r[j - l_i]
                if p > max
                    max <- p
        r[j] <- max
```

0-1 Knapsack problem: Thief robbing store finds m items worth $v_1, v_2, ..., v_m$ dollars and weight $w_1, w_2, ..., w_m$ lbs, where v_i, w_i are ints. Thief can carry at most W lbs. Which items should thief take to maximize value? Repetition is ok!

1a. Proof by contradiction. $\{i_1, i_2, ..., i_k\}$ is a solution to W but $\{i_2, ..., i_k\}$ isn't a solution to W - w_{i_1}. Then some solution to W - w_{i_1} exists where sum of value of items is greater than $\{i_2, ..., i_k\}$. This is a contradiction since we could use this to make a solution to the original problem that has more value.
1b. K(w) = $max_{i:w_i \leq w}\{K(w - w_i) + v_i\}$
2. 1d table. Need to solve subproblems 0...w-1. Smallest subproblem is K(0) = 0. Fill in from small to large.
3. space = $\Theta(W)$, time = $\Theta(Wm)$

Memoization: Sometimes challenging writing function bottom-up. Can Use memoization! Write recursive function top-down and alter to check if value is calculated. If yes, use pre-calculated value; do recursive call(s) otherwise.

```
Fibonacci-Memoized(n)
    fib[1], fib[2] <- 1, 1
    for i <- 3 to n
        fib[i] <- infinity
    return Fib-Lookup(n)

Fib-Lookup(n)
    if fib[n] < infinity
        return fib[n]
    x <- Fib-Lookup(n - 1) +
         Fib-Lookup(n - 2)
    if x < fib[n]
        fib[n] <- x
    return fib[n]
```

Levenshtein Distance: Edit distance between two strings is min number of insertions, deletions and substitutions required to transform s_1 into s_2.

$$Edit(X, Y) = \begin{cases} 1 + Edit(X_{1...n}, Y_{1...m-1}) & insertion \\ 1 + Edit(X_{1...n-1}, Y_{1...m}) & deletion \\ Diff(X_n, Y_m) + Edit(X_{1...n-1}, Y_{1...m-1}) & equal/sub \end{cases}$$

```
Edit(X, Y)
    m, n <- length[X], length[Y]
    for i <- 0 to m
        d[i, 0] <- i
    for j <- 0 to n
        d[0, j] <- j
    for i <- 1 to m
        for j <- 1 to n
            d[i, j] <- min(1 + d[i - 1, j],
                           1 + d[i, j - 1],
                           Diff(x_i, y_j) + d[i - 1, j - 1])
```

space = Θ(nm), time = Θ(nm)

Tree Induction 1: Show that a binary tree of height n has at most 2n leaf nodes.

- (base case, n = 0) if n = 0, tree is a single node so 1 leaf node which is $\leq$ 2.
- (inductive case, n = k' + 1) WTP a tree of height k' + 1 has at most 2^n leaf nodes. IH is that a tree of height n $\leq$ k' has at most $2^{k'}$ leaf nodes. A tree of height k' + 1 has two children that have height $\leq$ k'. Thus when we combine these two subtrees, we get that the total number of leaf nodes is at most $2^{k'} + 2^{k'} = 2^{k+1}$.

Tree Induction 2: Prove by strong induction on height of tree that a 1off tree with height h has at least f(h) nodes, where f(h) is h^{th} Fibonacci number

- (base cases, h = 0, h = 1) f(0) = 1 and height 0 is single node. f(1) = 1 and height 1 has 2 or 3 nodes so true.
- (inductive case, h = k' + 1) WTP a 1off tree with height k' + 1 has at least f(k' + 1) nodes. IH is that $\forall$ n $\leq$ k', a 1off tree with height n has at least f(n) nodes. Our tree has a root and two subtrees. At least one subtree has height k' and other one has height k' or k' - 1. Thus, tree has at least f(k') + f(k' - 1) + 1 nodes which is bigger than f(k' + 1).

From previous part, it's easily provable that 1off trees with height h have at least 2f(h - 2) nodes. We continue: n $\geq$ 2f(h-2) $\geq$ 4f(h-4) $\geq$ 8f(h-6) and so on. Eventually, height is h/2 when we reach f(1). We multiply by $2^{h/2}$ at this point, so n $\geq$ $2^{h/2}$. Upon rewriting, we get: 2log $n \geq$ h $\therefore$ n node 1off tree has $O(\log n)$ height.

Steps: A student is walking up a staircase with n steps. They can skip up to 2 steps at a time. Let Num(i) be the number of ways that the student can get to step i. Calculate Num(i).

Num(1) = 1, Num(2) = 2, Num(i) = Num(i - 1) + Num(i - 2). To get to step i, we can either take one step, which leaves us with i - 1 stairs left of we can take a bigger step, leaving i - 2 stairs left. Nearly identical to Fibonacci, fill in the first two values and then from 3 up to n. O(n) space and time.

Greedy algorithms: Algorithm that makes a local decision with goal of creating a globally optimal solution. Method for solving problems where optimal solutions can be defined in terms of optimal solutions to sub-problems.
Proving Greedy Algorithms Correct: One approach is to prove that (1) Optimal substructure: The optimal solution contains within it the optimal solution to sub problems and (2) Greedy choice property: The greedy choice is contained within some optimal solution.
Divide and Conquer vs Dynamic vs Greedy: With divide and conquer, the solution to the general problem is solved with respect to solutions to sub-problems. With DP, the solution to the general problem is solved with respect to solutions to sub-problems that overlap! With greedy, the solution to the general problem is solved by making locally optimal solutions.
Change problem: Input is number k and output is $\{n_p, n_n, n_d, n_q\}$ where $n_p + 5n_n + 10n_d + 25n_q = k$ and $n_p + n_n + n_d + n_q$ is minimized. To minimize, we pick as many quarters as we can, then dimes, then nickels, then pennies. Proof of correctness:

1. Optimal Substructure: We want to prove that if $\{c_1, c_2, c_3,..., c_m\}$ is optimal for k, then $\{c_2, c_3,...,c_m\}$ is optimal for k - c_1. We begin by assuming this is not the case. This would imply that there is some other set of coins $\{c'_2, c'_3,...,c'_m\}$ where $n < m$ that add up to k - c_1. If this were the case, $\{c_1, c'_2, c'_3,...,c'_m\}$ would be a solution to our original problem, but since $n < m$ this implies that our original solution wasn't optimal. Contradiction!
2. Greedy Choice Property: Want to prove that greedy choice results in optimal solution. Let $\{c_1, c_2, c_3,...,c_m\}$ be an optimal solution. Assume it is ordered from smallest to largest. Assume it does not make the greedy choice at some coin c_i. We need at least one more lower denomination coin because the greedy choice can be made up of c_i and one or more of the other denominations. This would mean the solution is longer than greedy, so contradiction!

Interval Scheduling: Given n activities A = $[a_1,a_2,...,a_n]$ where each activity has start time s_i and a finish time f_i. Schedule as many as possible of these activities such that they don't conflict.

- **Recursive & DP Approaches:** The simple recursive solution would enumerate all possible solutions and then find which schedules the most activities. The running time would be O(n!). With DP, we could get down to O(n^2). With greedy, we can do even better.
- **Greedy Approach:** Greedily pick an activity to schedule by picking the one that ends earliest. Add that activity to the answer. Remove that activity and all conflicting activities. Call this A'. Repeat on A' until A' is empty.
- **Stays ahead argument:** Show that no matter what other solution someone provides you, the solution provided by your algorithm always "stays ahead", in that no other choice could do better
- **Is this greedy approach correct?** Let $\{r_1, r_2, r_3,...,r_k\}$ be the solution found by our approach. Let $\{o_1, o_2, o_3,...,o_k\}$ be another optimal solution. We want to show that our approach "stays ahead" of any other solution. We start by comparing the first activities of each solution. We know that finish(r_1) $\leq$ finish(o_1). This implies that we have at least as much time as any other solution to schedule the remaining 2...k tasks.
- **Implementation:** To implement our solution efficiently, we need to sort by finish times and then iterate over all of the items—this would give us a run time of $\Theta(n \log n) + \Theta(n) = \Theta(n \log n)$.

Scheduling all activities: Given n activities, we need to schedule all activities. Goal: minimize the number of resources required.

- The best we can ever do is the maximum number of conflicts for any time period. We essentially need to move a hand across all of the activities and the maximum number under the hand at any time would be our answer. To do this, we would have to sort all of our start and end times and the iterate over them. Our time complexity would end up being O($n \log n$).

Minimum Spanning Trees: The lowest weight set of edges that connects all vertices of an undirected graph with positive weights. Input: An undirected, positive weight graph, G=(V,E). Output: A tree T=(V,E'), where E' $\subseteq$ E that minimizes weight(T) = $\sum_{e \in E'} w_e$. MSTs don't have cycles and have exactly $|V| - 1$ edges.
Cuts: A cut is a partitioning of the vertices into two sets S and V-S. An edge "crosses" the cut if it connects a vertex u $\in$ V and v $\in$ V-S.
Minimum Cut Property: Given a partition S, let edge e be the minimum cost edge that crosses the partition. Every minimum spanning tree contains edge e. To prove this, let us consider a MST with edge e' that is not the minimum edge. Using e instead of e', still connects the graph, but produces a tree with smaller weights. Caveat: if the minimum cost edge that crosses the partition is not unique, then some minimum spanning tree contains edge e.
Kruskal's Algorithm: Given a partition S, let edge e be the minimum cost edge that crosses the partition. Every minimum spanning tree contains edge e. Kruskal's works by adding the smallest edge that connects two sets not already connected.

```
Kruskal(G)
    for all v in V
        MakeSet(v)
    sort the edges of E by weight
    for all edges (u, v) in E in increasing order of weight
        if FindSet(u) != FindSet(v)
            add edge to T
            Union(FindSet(u), FindSet(v))
```

Correctness of Kruskal's: Never adds an edge that connects already connected vertices. Always adds lowest cost edge to connect two sets. By min cut property, that edge must be part of the MST.
Running time of Kruskal's: We call MakeSet $|V|$ times. We can sort the edges in $O(|E| \log |E|)$. We make $2|E|$ calls to FindSet and $|V|$ calls to union. If we assume that we are using a linked list to represent our disjoint set, MakeSet is constant, FindSet takes is $O(n)$ and Union is constant. This gives us $O(|V|) + O(|E| \log |E|) + O(|V||E|) + O(|V|) = O(|V||E|)$. If we use linked list + heuristics, we can get this down to $O(|V|) + O(|E| \log |E|) + O(\log |V||E|) + O(|V|) = O(|E| \log |E|)$
Prim's Algorithm: Start at some root node and build out the MST by adding lowest weighted edge at the frontier.

```
Prim(G, r)
    for all v in V
        key[v] <- infinity
        prev[v] <- null
    key[r] <- 0
    H <- MakeHeap(key)
    while !empty(H)
        u <- ExtractMin(H)
        visited[u] <- true
        for each edge (u, v) in E
            if !visited[v] and w(u, v) < key[v]
                DecreaseKey(v, w(u, v))
                prev[v] <- u
```

Correctness of Prim's: To prove, we use the min-cut property. Let S be the set of vertices visited so far. The only time we add a new edge is if it's the lowest weight edge from S to V-S. Thus, since for every edge added the min-cut property holds and we visit every node, we must have a MST by the end.
Running time of Prim's: The first loop takes $\Theta(|V|)$ time. We make one call to MakeHeap. We make $|V|$ calls to ExtractMin and we make $|E|$ calls to DecreaseKey. If our heap is implemented using an array, we need $\Theta(|V|) + O(|V|^2) + O(|E|) = O(|V|^2)$. If our heap is a binary heap, we need $\Theta(|V|) + O(|V| \log |V|) + O(|E| \log |V|) = O(|E| \log |V|)$.
DAGs: Directed Acyclic Graphs. Can represent dependency graphs.
Topological Sort: A linear ordering of all the vertices such that for all edges $(u, v) \in E$, u appears before v in the ordering. An ordering of the nodes that "obeys" the dependencies, i.e. an activity can't happen until it's dependent activities have happened.

```
Topological-Sort1(G)
    Find a node v with no incoming edges
    Delete Node v from G.
    Add v to linked list
    Topological-Sort1(G)
```

Running time: Finding the node without incoming edges takes $O(|V| + |E|)$. Deleting the node takes $O(|E|)$. We make $|V|$ calls. Thus, the running time is $O(|V|^2 + |E||V|)$.

```
Topological-Sort2(G)
    for all edges (u, v) in E
        active[v] <- active[v] + 1
    for all v in V
        if active[v] = 0
            Enqueue(S, v)
    while !Empty(S)
        u <- Dequeue(S)
        add u to linked list
        for each edge (u, v) in E
            active[v] <- active[v] - 1
            if active[v] = 0
                Enqueue(S, v)
```

Running time: $O(|V| + |E|)$.
Detecting Cycles: Undirected graph: BFS or DFS. If we reach a node we've seen already, then we've found a cycle. Directed graph: Call TopologicalSort and if the length of the list returned $\neq |V|$ then a cycle exists.
Connectedness: Given an undirected graph, for every node u $\in$ V, can we reach all other nodes in the graph? Run BFS or DFS-Visit (one pass) and mark nodes as we visit them. If we visit all nodes, return true, otherwise false. Running time: $O(|V| + |E|)$.
Transpose of a Graph: Given a graph G, we can calculate the transpose of a graph G^R by reversing the direction of all the edges. Running time to calculate G^R: $O(|V| + |E|)$.
Strongly Connected: Given a directed graph, can we reach any node v from any other node u?

```
Strongly-Connected(G)
    Run DFS-Visit or BFS from some node u
    If not all nodes are visited: return false
    Create graph G^R
    Run DFS-Visit or BFS on G^R from node u
    If not all nodes are visited: return false
    return true
```

Correctness: After first pass, we know that starting at u, we can reach every node. After the second pass, we know that every node can reach u. This means that any node can reach any other node. Given any two nodes s and t we can create a path through u.
Running time: DFS/BFS takes $O(|V| + |E|)$. Checking if nodes are visited takes $O(|V|)$. Creating graph G^R takes $O(|V| + |E|)$. Thus overall, we need $O(|V| + |E|)$.
Dijkstra's Algorithm: Find single-source shortest-path to all nodes on positive weighted graph.

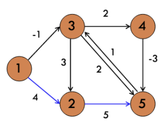
```
Dijkstra(G, s)
    for all v in V
        dist[v] <- infinity
        prev[v] <- null
    dist[s] <- 0
    Q <- MakeHeap(V)
    while !Empty(Q)
        u <- ExtractMin(Q)
        for all edges (u, v) in E
            if dist[v] > dist[u] + w(u, v)
                dist[v] <- dist[u] + w(u, v)
                DecreaseKey(Q, v, dist[v])
                prev[v] <- u
```

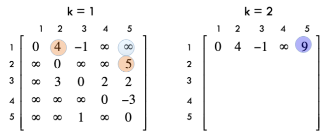
Correctness of Dijkstra's: Invariant: For every vertex removed from the heap, dist[v] is the actual shortest distance from s to v. The only time a vertex gets visited is when the distance from s to that vertex is smaller than the distance to any remaining vertex. Therefore, there cannot be any other path that hasn't been visited already that would result in a shorter path.
Running time: Depends on how the heap is implemented. If array, MakeHeap would be $O(|V|)$. The $|V|$ ExtractMin operations would be $O(|V|^2)$ and the $|E|$ DecreaseKey operations would be $O(|E|)$. This would give us an overall running time of $O(|V|^2)$. If binary heap, MakeHeap would be $O(|V|)$. The $|V|$ ExtractMin operations would be $O(|V| \log |V|)$ and the $|E|$ DecreaseKey operations would be $O(|E| \log |V|)$. This would give us an overall running time of $O(|E| \log |V|)$.
Update Rule Invariant: For each vertex v, dist[v] is an upper bound on the actual shortest distance. We start each vertex at ∞. Only update the value if we find a shorter distance. The update rule: $dist[v] = min\{dist[v], dist[u] + w(u, v)\}$. We can apply this rule as many times as we want and will never underestimate dist[v]. When will dist[v] be right? If u is along the shortest path to v and dist[u] is correct. If we apply the update rule $|V| - 1$ times, we will guarantee that we have included the longest shortest path.
Bellman-Ford Algorithm:

```
Bellman-Ford(G, s)
    for all v in V
        dist[v] <- infinity
        prev[v] <- null
    dist[s] <- 0
    for i <- 1 to |V| - 1
        for all edges (u, v) in E
            if dist[v] > dist[u] + w(u, v)
                dist[v] <- dist[u] + w(u, v)
                prev[v] <- u
    for all edges (u, v) in E
        if dist[v] > dist[u] + w(u, v)
            return false
```

Correctness of Bellman-Ford: Lop invariant: After iteration i, all vertices with shortest paths from s of length i edges or less have correct distances.
Running time of Bellman-Ford: $O(|V||E|)$
All Pairs Shortest Paths: calculate the shortest paths between all vertices. The easy solution would be to run Bellman-Ford from every vertex—the running time of this would be $O(|E||V|^2)$ since $|V|$ calls (one for each vertex).
Floyd-Warshall: Label all vertices with a number 1 to $|V|$ and d_{ij}^k is the shortest path from vertex i to vertex j using only vertices $\{1, 2, 3, ..., k\}$. There are $|V|^3$ possible values for d_{ij}^k since i is all vertices, j is all vertices and k is all vertices. The value of d_{ij}^V is simply the shortest path from i to j—if we have this for all i,j, then we're done.
Recursive Relationship: Assume we know d_{ij}^k. How can we compute d_{ij}^{k+1} i.e. shortest path now including vertex k + 1 (in terms of d_{ij}^k). There are two options: vertex v + 1 gives us a shorter path or it doesn't. If it doesn't then, $d_{ij}^{k+1} = d_{ij}^k$. If it does, then $d_{ij}^{k+1} = d_{i(k+1)}^k + d_{(k+1)j}^k$. To combine these two options, we say: $d_{ij}^{k+1} = \min\{d_{ij}^k, d_{i(k+1)}^k + d_{(k+1)j}^k\}$

```
Floyd-Warshall(G = (V, E, W))
    initialize d with edge weights
    for k <- 1 to K
        for i <- 1 to V
            for j <- 1 to V
                d_ij^k = min(d_ij^k-1, d_ik^k-1 + d_kj^k-1)
    return d
```

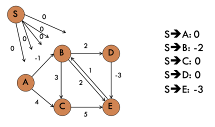


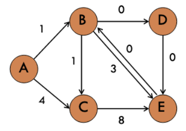


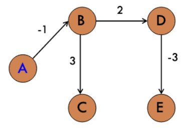
$$k = 1: \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 4 & -1 & \infty & \infty \\ 2 & \infty & 0 & \infty & \infty & 5 \\ 3 & \infty & 3 & 0 & 2 & 2 \\ 4 & \infty & \infty & \infty & 0 & -3 \\ 5 & \infty & \infty & 1 & \infty & 0 \end{array} \qquad k = 2: \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 4 & -1 & \infty & 9 \\ 2 & & & & & \\ 3 & & & & & \\ 4 & & & & & \\ 5 & & & & & \end{array}$$

Correctness of Floyd-Warshall: The algorithm is correct as long as there are no negative cycles. If the graph has a negative weight cycle, at the end, at least one of the diagonal entries will be a negative number, i.e., we there's a way to get back to a vertex using all of the vertices that results in a negative weight.
Analysis of Floyd-Warshall: Running time is $O(|V|^3)$ and space usage is $O(|V|^2)$
Dijkstra's for all pairs shortest path: if the edge weights are positive, we can calculate all pairs by running Dijkstra's from each vertex. Since Dijkstra's has a running time of $O(|V| \log |V| + |E|)$, this would have a running time of $O(|V|^2 \log |V| + |E||V|)$. This is better than Floyd-Warshall if the graph is sparse.
Johnson's Algorithm: The challenge with doing the $|V|$ calls of Dijkstra's is that edge weights need to be positive. For Johnson's, the key idea is re-weighting the edges such that the shortest paths are preserved. To do this, we use the following lemma: $\hat{w}(u, v) = w(u, v) + h(u)$ - $h(v)$ where h is any function mapping a vertex to a real value. We need to pick h such that all the edge weights end up as positive.
Lemma Proof: Claim: the weight change preserves shortest paths, i.e. if a path was the shortest from s to t in the original graph it will still be the shortest path from s to t in the new graph. Proof: h(s) – h(t) is a constant and will be the same for all paths from s to t, so the absolute ordering of all paths from s to t will not change.

```
Johnsons(G)
    Create G' with one extra node s with 0 weight edges to all nodes
    run Bellman-Ford(G',s)
    if no negative weight cycle
        reweight edges in G with h(v)=shortest path from s to v
        run Dijkstra's from every vertex
        reweight shortest paths based on G
```







Running time of Johnson's: Creating G' takes $O(|V|)$. Running Bellman-Ford takes $O(|V|^2)$. Reweighting the edges take $O(|E|)$. Running Dijkstra's from every vertex takes $O(|V|^2 \log |V| + |E||V|)$. Reweighting the shortest paths takes $O(|E|)$. Thus the overall running time is $O(|V|^2 \log |V| + |E||V|)$.
How can we guarantee positive edge weights? Take two nodes u and v. h(u) = shortest distance from s to u. h(v) = shortest distance from s to v. We claim that h(v) $\leq$ h(u) + w(u, v). If this weren't true, we could have made a shorter path s to v using u but this is in contradiction with how we defined h(v). Next, we rewrite the inequality as w(u, v) + h(u) - h(v) $\geq$ 0. Thus, all weights are guaranteed to be positive!
A7 Greedy Problem: On any given day you can drive at most d miles. You have a map with n different hotels and the distances from your start point to each hotel $x_1 < x_2 < ... < x_n$. Your final destination is the last hotel. Describe an algorithm that determines which hotels you should stay in if you want to minimize the number of days it takes you to get to your destination.

Greedy algorithm that picks the furthest hotel that is d miles or less away. Next, it would pick the furthest hotel that is d miles or less away from that hotel. Continue doing this until we reach our final destination. Our algorithm would run in $O(n)$ time (we would consider two hotels at a time to see if we are at a threshold where the next hotel is too far away from the previous hotel we stayed at). To prove our solution is optimal, we use proof by contradiction. Since we can't go more than d miles in a day, assume that we go to a closer hotel than described. Then we can get no further than we would have been staying in the furthest possible hotel, and this pattern repeats for all subsequent days. Therefore, we cannot arrive at our destination any earlier. Thus, following the pattern we described will always produce an optimal solution, though there may be other optimal solutions.
A8 Graph Problems:

1a. Prim's algorithm still works with negative edge weights since the min-cut property still holds.
1b. The shortest path between two nodes isn't necessarily part of the MST. Prove with counterexample.

A9 Graph Problems:

1. Given an undirected graph G and an edge e = (u, v), determine whether G has a cycle containing e. Solution: remove e from the graph. Run BFS on this new graph starting at u marking each vertex as visited. If we visit v, then there is a cycle in the graph with edge (u, v). Running time = $O(V + E)$.
2. Given an undirected, unweighted graph G and nodes $u, v \in V$, output the number of distinct shortest paths from u to v. Solution: Modify BFS. For each vertex keep track of the number of paths possible to that vertex, call that count. Set count[v] = 0 for all vertices. Set count[s] = 1 for the starting vertex. When exploring a new edge, if dist[v] > dist[u] + w(u, v), set count[v] = count[u]. If dist[v] = dist[u] + w(u, v), set count[v] += count[u].
3. Given a directed graph G = (V, E) with positive edge weights and a particular node $v_i \in V$, give an efficient algorithm for finding the shortest paths between all pairs of nodes, with the one restriction that these paths must all pass through v_i. Solution: Run Dijkstra's starting at v_i on G. Calculate G^R and run Dijkstra's starting at v_i on G^R. The shortest path from any vertices a to b through v_i is the shortest path from v_i to a in G^R combined with the shortest path from v_i to b in G. The running time is bounded by two calls to Dijkstra's: $O(V \log V + E)$.
4. If a graph has no negative cycle, when calculating the shortest paths from a given vertex using Bellman-Ford, we can stop early and do not need to do all $|V| - 1$ iterations and will still have a correct answer for all the shortest paths from that vertex. Describe how to modify the Bellman-Ford algorithm to stop early when all of the distances are already correct. Solution: If we make a pass through all of the edges and don't update any of the dist value i.e. we don't find a case where dist[v] > dist[u] + w(u, v), then we've found all of the shortest paths and can stop early.