14.2-2
Give a recursive algorithm MATRIX-CHAIN-MULTIPLY(A, s, i, j) that actually performs the optimal matrix-chain multiplication, given the sequence of matrices $\langle A_1, A_2, \ldots, A_n \rangle$, the s table computed by MATRIX-CHAIN-ORDER, and the indices i and j. (The initial call is MATRIX-CHAIN-MULTIPLY$(A, s, 1, n)$.) Assume that the call RECTANGULAR-MATRIX-MULTIPLY(A, B) returns the product of matrices A and B.

```
MATRIX-CHAIN-MULTIPLY(A, s, i, j)
1. if i == j
2.     return A[i]
3. else
4.     X ← MATRIX-CHAIN-MULTIPLY(A, s, i, s[i][j])
5.     Y ← MATRIX-CHAIN-MULTIPLY(A, s, s[i][j] + 1, j)
6.     return RECTANGULAR-MATRIX-MULTIPLY(X, Y)
```

14.2-4
Describe the subproblem graph for matrix-chain multiplication with an input chain of length n. How many vertices does it have? How many edges does it have, and which edges are they?

14.2-4: Subproblem Graph for Matrix-Chain Multiplication

- **Vertices (Subproblems** $m[i, j]$ **where** $i < j$**)**

$$\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} \Rightarrow \boxed{\text{Vertices} = \frac{n(n-1)}{2}}$$

- **Edges (Each** $m[i, j]$ **depends on** $m[i, k]$ **and** $m[k+1, j]$**)**

Each pair (i, j) has $j - i$ values of $k \Rightarrow 2(j - i)$ edges

$$\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 2(j-i) = 2\sum_{d=1}^{n-1}\sum_{i=1}^{n-d} d = 2\sum_{d=1}^{n-1}\frac{(n-i)(n-i+1)}{2} = \Theta(n^3) \quad \Rightarrow \boxed{\text{Edges} = \Theta(n^3)}$$

- **Edge Form:**

$$(i, j) \to (i, k) \quad \text{and} \quad (i, j) \to (k+1, j)$$

$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$

14.2-3
Use the substitution method to show that the solution to the recurrence (14.6) is $\Omega(2^n)$.

Step 1: Guess

Assume $P(n) \geq c \cdot 2^n$ for some constant $c > 0$

Step 2: Base Case

For $n = 1$.

$$P(1) = 1 \geq c \cdot 2^1 = 2c \Rightarrow c \leq \frac{1}{2}$$

Choose $c = \frac{1}{2}$

Step 3: Inductive Hypothesis

Assume $P(m) \geq c \cdot 2^m$ for all $m < n$

Step 4: Inductive Step

$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k) \geq \sum_{k=1}^{n-1} c \cdot 2^k \cdot c \cdot 2^{n-k} = c^2 \cdot \sum_{k=1}^{n-1} 2^k \cdot 2^{n-k} = c^2 \cdot \sum_{k=1}^{n-1} 2^n = c^2(n-1)2^n$$

To ensure $P(n) \geq c \cdot 2^n$:

$$c^2(n-1)2^n \geq c \cdot 2^n \Rightarrow c(n-1) \geq 1 \Rightarrow n \geq \frac{1}{c} + 1$$

If $c = \frac{1}{2}$, then $n \geq 3$

Final Result

$$\boxed{P(n) = \Omega(2^n)} \quad \text{(holds for all } n \geq 3\text{)}$$

14.2-5
Let $R(i, j)$ be the number of times that table entry $m[i, j]$ is referenced while computing other table entries in a call of MATRIX-CHAIN-ORDER. Show that the total number of references for the entire table is

$$\sum_{i=1}^{n}\sum_{j=i}^{n} R(i, j) = \sum_{i=1}^{n}\sum_{j=i}^{n} 2(n - i + 1)(i - 1)$$
$$= 2\sum_{i=1}^{n}(n-i)i$$
$$= 2\sum_{i=0}^{n-1} nl - 2\sum_{i=0}^{n-1} l^2$$
$$= \frac{n(n-1)n}{2} - \frac{(n-1)n(2n-1)}{6}$$
$$= n^3 - n^2 - \frac{2n^3 - 3n^2 + n}{3}$$
$$= \frac{n^3 - n}{3}.$$

Solution to Exercise 14.2-5

Each time the l-loop executes, the i-loop executes $n - l + 1$ times. Each time the i-loop executes, the k-loop executes $j - i = l - 1$ times. each time referencing m twice. Thus the total number of times that an entry of m is referenced while computing other entries is $\sum_{l=2}^{n} 2(n - l + 1)(l - 1)$. Thus,

14.2-6
Show that a full parenthesization of an n-element expression has exactly $n-1$ pairs of parentheses.

Proof (by induction):

- **Base Case:**

For $n = 2$:
Expression: $(A_1 A_2) \rightarrow 1$ **pair** of parentheses

$$= 2 - 1 = 1 \quad \text{holds}$$

- **Inductive Hypothesis:**

Assume true for all expressions of size $k < n$:
A full parenthesization of k elements has $k - 1$ pairs

- **Inductive Step:**

Split the expression into two parts:

- Left: i elements → needs $i - 1$ pairs
- Right: $n - i$ elements → needs $(n - i) - 1$ pairs
- Add 1 pair for combining them

$$\text{Total} = (i - 1) + (n - i - 1) + 1 = n - 1 \quad \text{proven}$$

14.5-1
Give an $O(n^2)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers.

```
LIS-LENGTH(A[1..n])
    let LIS[1..n] be an array initialized to 1

    for i = 2 to n:
        for j = 1 to i-1:
            if A[j] < A[i]:
                LIS[i] = max(LIS[i], LIS[j] + 1)

    return max over LIS[1..n]
```

Show how to compute the length of an LCS using only $2 \cdot \min\{m, n\}$ entries in the c table plus $O(1)$ additional space. Then show how to do the same thing, but using $\min\{m, n\}$ entries plus $O(1)$ additional space.

```
Let m = length(X), n = length(Y)

Assume n <= m   // if not, swap X and Y to minimize space

Create two arrays: prev[0..n], curr[0..n]

Initialize prev[0..n] = 0

for i = 1 to m:
    curr[0] = 0
    for j = 1 to n:
        if X[i] == Y[j]:
            curr[j] = prev[j-1] + 1
        else:
            curr[j] = max(prev[j], curr[j-1])
    swap curr and prev

return prev[n]
Let m = length(X), n = length(Y)
Assume n <= m   // again, swap if necessary

Create array dp[0..n] and initialize to 0

for i = 1 to m:
    prev = 0
    for j = 1 to n:
        temp = dp[j]        // stores old dp[j] for use in next iteration
        if X[i] == Y[j]:
            dp[j] = prev + 1
        else:
            dp[j] = max(dp[j], dp[j-1])
        prev = temp

return dp[n]
```

Give a memoized version of LCS-LENGTH that runs in $O(mn)$ time.

```
MEMOIZED-LCS-LENGTH(X, Y)
    m = length(X)
    n = length(Y)
    let c[0..m][0..n] be a table filled with -1  // -1 means "not computed"

    return LCS-LOOKUP(X, Y, m, n, c)

LCS-LOOKUP(X, Y, i, j, c)
    if i == 0 or j == 0
        return 0

    if c[i][j] ≠ -1
        return c[i][j]

    if X[i] == Y[j]
        c[i][j] = LCS-LOOKUP(X, Y, i-1, j-1, c) + 1
    else
        c[i][j] = max(
            LCS-LOOKUP(X, Y, i-1, j, c),
            LCS-LOOKUP(X, Y, i, j-1, c)
        )

    return c[i][j]
```

Give an $O(\lg n)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers. (*Hint:* The last element of a candidate subsequence of length i is at least as large as the last element of a candidate subsequence of length $i - 1$. Maintain candidate subsequences by linking them through the input sequence.)

```
LIS-LENGTH(A[1..n])
    tail = empty list

    for x in A:
        pos = LOWER-BOUND(tail, x)   // binary search: first index with tail[pos] ≥ x
        if pos == length(tail):
            tail.append(x)
        else:
            tail[pos] = x

    return length(tail)
```

20.4-2
Give a linear-time algorithm that, given a directed acyclic graph $G = (V, E)$ and two vertices s and $t \in V$, returns the number of simple paths from s to t in G. For example, the directed acyclic graph of Figure 20.8 contains exactly four simple paths from vertex p to vertex v: $\langle p, o, v \rangle$, $\langle p, o, r, y, v \rangle$, $\langle p, o, s, r, y, v \rangle$, and $\langle p, s, r, y, v \rangle$. Your algorithm needs only to count the simple paths, not list them.

```
COUNT-SIMPLE-PATHS(G, a, b):
1. top_order ← TOPOLOGICAL-SORT(G)
2. for each v in V:
3.     paths[v] ← 0
4. paths[a] ← 1
5. for each u in top_order:
6.     for each (u, v) in G.adj[u]:
7.         paths[v] ← paths[v] + paths[u]
8. return paths[b]
```

20.4-3
Give an algorithm that determines whether an undirected graph $G = (V, E)$ contains a simple cycle. Your algorithm should run in $O(V)$ time, independent of $|E|$.

```
HAS-SIMPLE-CYCLE(G):
1. count ← 0
2. for each vertex v in V:
3.     count ← count + 1
4. if |E| ≥ count:
5.     return TRUE  // cycle exists
6. else:
7.     return FALSE
```

20.4-4
Prove or disprove: If a directed graph G contains cycles, then the vertex ordering produced by TOPOLOGICAL-SORT(G) minimizes the number of "bad" edges that are inconsistent with the ordering produced.

Disproof (Counterexample):

- Topological sort is only defined for **DAGs**
- If G has a **cycle**, no valid topological order exists
- Any ordering is arbitrary and does not minimize back edges

Example:

Cycle: $A \to B \to C \to A$
Any order gives at least one "bad" (backward) edge

Final Verdict:

False: Topological sort cannot be applied to cyclic graphs, so it can't minimize bad edges.

20.5-1
How can the number of strongly connected components of a graph change if a new edge is added?

- Adding a new edge **can decrease** the number of strongly connected components (SCCs)
- It **can never increase** the number of SCCs
- If the new edge **connects two different SCCs bidirectionally**, they may merge into one

20.5-2
Professor Bacon rewrites the algorithm for strongly connected components to use the original (instead of the transpose) graph in the second depth-first search and scan the vertices in order of *increasing* finish times. Does this modified algorithm always produce correct results?

- Kosaraju's SCC algorithm **requires** the second DFS to run on the **transpose** of the graph and in **decreasing** order of finish times
- Professor Bacon:
  - Uses the **original** graph instead of the transpose
  - Uses **increasing** finish times instead of decreasing

Both changes break the correctness:

- Strong connectivity is not preserved in original graph traversal
- The finish-time order is crucial to isolate SCC roots correctly

20.5-3
Professor Bacon rewrites the algorithm for strongly connected components to use the original (instead of the transpose) graph in the second depth-first search and scan the vertices in order of *increasing* finish times. Does this modified algorithm always produce correct results?

- Kosaraju's SCC algorithm **requires** the second DFS to run on the **transpose** of the graph and in **decreasing** order of finish times
- Professor Bacon:
  - Uses the **original** graph instead of the transpose
  - Uses **increasing** finish times instead of decreasing

Both changes break the correctness:

- Strong connectivity is not preserved in original graph traversal
- The finish-time order is crucial to isolate SCC roots correctly

20.5-4
Prove that for any directed graph G, the transpose of the component graph of G^{SCC} is the same as the component graph of G. That is, $((G^{\text{SCC}})^{\text{T}})^{\text{SCC}} = G^{\text{SCC}}$.

- Let $C_i \to C_j$ be an edge in G^{SCC}
  - ⇒ There exists an edge $u \to v$ in G such that $u \in C_i, v \in C_j$
- In G^{T}, we have edge $v \to u$
  - ⇒ So $C_j \to C_i$ is in $(G^{\text{T}})^{\text{SCC}}$
- Taking the transpose again gives $C_i \to C_j$

If G **has cycles:**

- Some nodes never reach in-degree 0
- Queue becomes empty early
- Not all nodes are output ⇒ **Cycle detected**

Final Answer:

Runs in $O(V + E)$; Fails (incomplete output) if G contains cycles.

20.5-5
Give an $O(V + E)$-time algorithm that, given a directed graph $G = (V, E)$, constructs another graph $G' = (V, E')$ such that G and G' have the same strongly connected components, G' has the same component graph as G, and $|E'|$ is as small as possible.

```
BUILD-MINIMAL-G'(G = (V, E)):

1. Run Kosaraju's or Tarjan's algorithm to label each vertex v with its scc_id[v]

2. Initialize G' = (V, E') with E' = ∅
3. Initialize empty set EdgeSet

4. For each edge (u, v) ∈ E:
5.     Cu ← scc_id[u], Cv ← scc_id[v]
6.     If Cu = Cv:
7.         Add (u, v) to E'   // Keep all intra-SCC edges
8.     Else if (Cu, Cv) ∉ EdgeSet:
9.         Add (u, v) to E'   // Add one inter-SCC edge
10.        Add (Cu, Cv) to EdgeSet

11. Return G' = (V, E')
```

20.5-6
Give an $O(V + E)$-time algorithm that, given a directed graph $G = (V, E)$ is semiconnected if, for all pairs of vertices $u, v \in V$, we have $u \rightsquigarrow v$ or $v \rightsquigarrow u$. Give an efficient algorithm to determine whether G is semiconnected. Prove that your algorithm is correct, and analyze its running time.

```
IS-SEMICONNECTED(G):
1. Run Kosaraju's or Tarjan's algorithm to find SCCs
2. Build component graph G_SCC (a DAG of SCCs)
3. topologically sort G_SCC → get order [C1, C2, ..., Ck]
4. For i = 1 to k-1:
5.     If no edge from Ci to Ci+1:
6.         return FALSE
7. return TRUE
```

24.5-2
Suppose that we redefine the residual network to disallow edges into s. Argue that the procedure FORD-FULKERSON still correctly computes a maximum flow.

Yes, **Ford-Fulkerson still correctly computes the maximum flow**, even if we disallow residual edges **into the source** s.

In the residual graph, a **reverse edge** (v, u) with capacity $f(u, v)$ exists to allow "undoing" flow.

However, the **source** s has **no incoming flow** in a valid flow (by definition).

Therefore, **reverse residual edges into** s are **never used** in augmenting paths.

24.5-3
Suppose that each source s_i in a flow network with multiple sources and sinks produces exactly p_i units of flow, so that $\sum_{v \in V} f(s_i, v) = p_i$. Suppose also that each sink t_j consumes exactly q_j units, so that $\sum_{v \in V} f(v, t_j) = q_j$, where $\sum_i p_i = \sum_j q_j$. Show how to convert the problem of finding a flow f that obeys these additional constraints into the problem of finding a maximum flow in a single-source, single-sink flow network.

Step 1: Add a super-source s'

- Add edges (s', s_i) for each original source s_i
- Set **capacity** $c(s', s_i) = p_i$
- This ensures that each source **sends exactly** p_i units — not just at most

Step 2: Add a super-sink t'

- Add edges (t_j, t') for each original sink t_j
- Set **capacity** $c(t_j, t') = q_j$
- This ensures each sink **receives exactly** q_i units

The construction in Section 24.1 to convert a flow network with multiple sources and sinks into a single-source, single-sink network adds edges with infinite capacity. Prove that any flow in the resulting network has a finite value if the edges of the original network with multiple sources and sinks have finite capacity.

1. All augmenting paths in the network must pass through original edges that have finite capacity.

- Even though the edges from s' to s_i and from t_j to t' have **infinite capacity**, these do not enable infinite flow on their own.
- **Flow conservation** still applies at intermediate nodes and all flow must be routed through the **original edges**, which **limit the total flow**.

2. Total flow is bounded above by the sum of capacities of the original edges:

$$\text{Maximum possible flow} \leq \sum_{(u,v) \in E} c(u, v) \quad \text{where } c(u, v) \text{ is finite}$$

3. Infinite capacity edges are only there to allow full use of the actual flow capacity.

- They **do not add** any additional capacity to the network.
- They just let you **aggregate** all the flows into a common source and sink.

Since all the actual bottlenecks (original edges) have finite capacity, and since the added infinite-capacity edges do not create new flow paths—just facilitate combining multiple sources and sinks—the total flow in the transformed network must be **finite**.

Suppose that both f and f' are flows in a flow network. Does the augmented flow $f \uparrow f'$ satisfy the flow conservation property? Does it satisfy the capacity constraint?

1. Flow Conservation

Yes, the augmented flow $f \uparrow f'$ satisfies the **flow conservation** property.

- At every node $v \neq s, t$, the net inflow equals net outflow in both f and f'.
- Therefore, the sum $f(u, v) + f'(u, v)$ still satisfies:

$$\sum_w (f + f')(u, v) = \sum_w (f + f')(v, w)$$

No, $f \uparrow f'$ **does not necessarily satisfy** the capacity constraint.

- The total flow on an edge:

$$(f \uparrow f')(u, v) = f(u, v) + f'(u, v)$$

may exceed the original edge capacity $c(u, v)$.

Show how to find a maximum flow in a flow network $G = (V, E)$ by a sequence of at most $|E|$ augmenting paths. (*Hint:* Determine the paths *after* finding the maximum flow.)

1. **Run any max-flow algorithm** (e.g., Ford-Fulkerson or Edmonds-Karp) to compute a maximum flow f
2. **Decompose the flow** f into **at most** $|E|$ augmenting paths (flow-carrying paths from source to sink)
   - Since each unit of flow on an edge contributes to total flow,
   - and each **positive flow edge** can only appear once per path,
   - there are **at most** $|E|$ such **nonzero-flow** edges.
3. For each **non-zero flow edge**, trace back a path from s to t in the **flow graph** and subtract flow accordingly.

Any maximum flow can be decomposed into a sequence of **at most** $|E|$ augmenting paths **after** computing it. This does not help reduce runtime of algorithms, but proves an upper bound on path decomposition.

The *edge connectivity* of an undirected graph is the minimum number k of edges that must be removed to disconnect the graph. For example, the edge connectivity of a tree is 1, and the edge connectivity of a cyclic chain of vertices is 2. Show how to determine the edge connectivity of an undirected graph $G = (V, E)$ by running a maximum-flow algorithm on at most $|V|$ flow networks, each having $O(V + E)$ vertices and $O(E)$ edges.

1. **Fix a vertex** $s \in V$ as the source.
2. For **each vertex** $t \in V \setminus \{s\}$:
   - Convert the **undirected graph** to a **directed flow network**:
     - Replace each undirected edge $\{u, v\}$ with **two directed edges** (u, v) and (v, u), each with **capacity = 1**.
   - Run a **max-flow algorithm** from s to t.
   - Let the flow value be f_{st}.
3. **Edge connectivity** $k = \min_{s \neq t} f_{st}$
   - By the **Max-Flow Min-Cut Theorem**, the minimum number of edges that must be removed to disconnect s and t equals the **max flow** from s to t.
   - Repeating this for all $t \neq s$ ensures you find the **global minimum** over all pairs.
   - You run at most $|V| - 1$ max-flow computations.
   - Each flow network has $O(V + E)$ size (since each edge becomes two directed edges).
   - Overall time depends on the max-flow algorithm used.

You are given a flow network G, where G contains edges entering the source s. Let f be a flow in G with $|f| \geq 0$ in which one of the edges (v, s) entering the source has $f(v, s) = 1$. Prove that there must exist another flow f' with $f'(v, s) = 0$ such that $|f| = |f'|$. Give an $O(E)$-time algorithm to compute f', given f and assuming that all edge capacities are integers.

1. Since f is a valid flow with $f(v, s) = 1$, this flow must **exit again** from s (by flow conservation).
2. So, there must be a **cycle** involving edge (v, s).
3. We can remove **1 unit of flow along this cycle**, yielding a new flow f' where $f'(v, s) = 0$, and total flow value is **unchanged**.
4. Build the **residual graph** of f.
5. **Find a path (cycle)** from s to v in the residual graph (since $f(v, s) = 1$, such a path must exist).
6. **Augment -1 unit** of flow along the cycle $v \to s \to \cdots \to v$.
7. Result is flow f' with:
   - $f'(v, s) = 0$
   - $|f'| = |f|$

This only takes $O(E)$ time using DFS or BFS in the residual graph.

Suppose that you wish to find, among all minimum cuts in a flow network G with integer capacities, one that contains the smallest number of edges. Show how to modify the capacities of G to create a new flow network G' in which any minimum cut in G' is a minimum cut with the smallest number of edges in G.
Let $C(u, v)$ be the original capacity of edge (u, v) in G.
Define the new capacity function $C'(u, v)$ for the modified graph G' as:

$$C'(u, v) = C(u, v) \cdot M + 1$$

Where:

- M is a large constant, **greater than the total number of edges**, e.g., $M = |E|$
- In this new network G', the **main term** $C(u, v) \cdot M$ dominates the cut capacity.
- The **+1 acts as a tie-breaker**: among cuts with equal total capacity (in original units), it favors those with **fewer edges**.
- So the minimum cut in G' will:
  - Minimize original capacity
  - Among those, **minimize edge count**

Suppose that you wish to find, among all minimum cuts in a flow network G with integer capacities, one that contains the smallest number of edges. Show how to modify the capacities of G to create a new flow network G' in which any minimum cut in G' is a minimum cut with the smallest number of edges in G.
Let $C(u, v)$ be the original capacity of edge (u, v) in G.
Define the new capacity function $C'(u, v)$ for the modified graph G' as:

$$C'(u, v) = C(u, v) \cdot M + 1$$

Where:

- M is a large constant, **greater than the total number of edges**, e.g., $M = |E|$
- In this new network G', the **main term** $C(u, v) \cdot M$ dominates the cut capacity.
- The **+1 acts as a tie-breaker**: among cuts with equal total capacity (in original units), it favors those with **fewer edges**.
- So the minimum cut in G' will:
  - Minimize original capacity
  - Among those, **minimize edge count**

To find a **minimum cut with the fewest edges**, create G' with:

$$C'(u, v) = C(u, v) \cdot (|E| + 1) + 1$$

Now run **any max-flow algorithm** on G' and extract the corresponding min cut.

Conclusion:

$$\boxed{((G^{\text{T}})^{\text{SCC}})^{\text{T}} = G^{\text{SCC}}}$$

Because reversing all edges **twice** returns the original component graph.

20.5-5
Give an $O(V + E)$-time algorithm to compute the component graph of a directed graph $G = (V, E)$. Make sure that there is at most one edge between two vertices in the component graph your algorithm produces.

```
BUILD-COMPONENT-GRAPH(G = (V, E)):

1. Run Kosaraju's or Tarjan's algorithm to compute SCCs
2. For each vertex v ∈ V:
3.     scc_id[v] ← component number of vertex v

4. Initialize CompGraph as an empty adjacency list of size = number of SCCs
5. Initialize EdgeSet as an empty hash set

6. For each edge (u, v) ∈ E:
7.     Cu ← scc_id[u]
8.     Cv ← scc_id[v]
9.     If Cu ≠ Cv and (Cu, Cv) ∉ EdgeSet:
10.        Add (Cu → Cv) to CompGraph
11.        Add (Cu, Cv) to EdgeSet

12. Return CompGraph
```

Let $G = (V, E)$ be a bipartite graph with vertex partition $V = L \cup R$, and let G' be its corresponding flow network. Give a good upper bound on the length of any augmenting path found in G' during the execution of FORD-FULKERSON.

In the flow network G', each augmenting path goes from:

$$s \to \ell \in L \to r \in R \to t$$

So the path includes:

- One edge from $s \to \ell$
- One edge from $\ell \to r$
- One edge from $r \to t$

$$\text{Length of any augmenting path} \leq 3$$

This bound holds throughout Ford-Fulkerson because:

- The flow network is built from a bipartite graph with edges only from $L \to R$
- The super source and sink only connect to one side of the partition

Prove that the fractional knapsack problem has the greedy-choice property. Assume there's an optimal solution that doesn't take items in greedy order.

Let:

- Item i has **higher** value density than item j
- Optimal solution takes **some of item** j but **not full of** i

Then:

- Replace some of j with the same weight of i
- This **increases total value**, as $\frac{v_i}{w_i} > \frac{v_j}{w_j}$

Contradiction: original solution wasn't optimal

Give a dynamic-programming solution to the 0-1 knapsack problem that runs in $O(nW)$ time, where n is the number of items and W is the maximum weight of items that the thief can put in the knapsack.

Let T be a minimum spanning tree of a graph $G = (V, E)$, and let V' be a subset of V. Let T' be the subgraph of T induced by V', and let G' be the subgraph of G induced by V'. Show that if T' is connected, then T' is a minimum spanning tree of G'.

1. $T' \subseteq T$, so all edge weights in T' are from T, i.e., minimal.
2. Suppose there exists another spanning tree T'' of G' with weight less than T'.
3. Then, replacing edges of T' with those in T'' (within V') in T would produce a spanning tree of G with **lesser total weight**.
4. ❌ This contradicts that T is an MST of G

Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Kruskal's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W?

- Use **Counting Sort** or **Bucket Sort**:
  - Time: $O(E + |V|)$
- Union-Find takes $O(E \cdot \alpha(V))$ (almost linear)

Total runtime:

$$O(E + V + E \cdot \alpha(V)) = O(E + V)$$

- Counting Sort now takes $O(E + W) = O(E)$ (since W is constant)
- Union-Find remains $O(E \cdot \alpha(V))$

Total runtime:

Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Prim's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W?

- With weights in $[1, |V|]$: Prim's algorithm runs in $O(E + V)$
- With weights in $[1, W]$ for constant W: it runs in $O(E)$ time

Kruskal's algorithm can return different spanning trees for the same input graph G, depending on how it breaks ties when the edges are sorted. Show that for each minimum spanning tree T of G, there is a way to sort the edges of G in Kruskal's algorithm so that the algorithm returns T.

1. Let T be a minimum spanning tree of G.
2. Since T has minimal total weight, Kruskal's algorithm **can** return it, provided it picks the **same edges as in** T when given the chance.
3. Modify the sorting of the edges so that:
   - All edges in T come **before** any edge **not in** T, when weights are equal.
4. Now run Kruskal's algorithm:
   - Since edges of T are sorted first, and no cycles form as we add them, Kruskal's algorithm will **exactly construct** T.

Therefore, for **any MST** T, there exists a **tie-breaking sorting** that ensures **Kruskal's algorithm returns** T

Suppose that a graph G has a minimum spanning tree already computed. How quickly can you update the minimum spanning tree upon adding a new vertex and incident edges to G?

T be the current MST of G

v be the **new vertex** added

k be the number of **edges from** v to existing vertices

To maintain the MST:

1. Find the **minimum-weight edge** (v, u) connecting v to any vertex in T
2. Add this edge to T

This is correct because:

- T already spans all original vertices
- Adding v via the lightest edge preserves minimality and acyclicity

Optimal update time: $\boxed{O(k)}$

Let (u, v) be a minimum-weight edge in a connected graph G. Show that (u, v) belongs to some minimum spanning tree of G.

- Consider the cut that separates vertex u from all other vertices.
- Since (u, v) is the **minimum-weight edge in the entire graph**, it is certainly the **lightest edge across this cut**.

Therefore, by the cut property, (u, v) **must belong to some MST** of G.

Show that if an edge (u, v) is contained in some minimum spanning tree, then it is a light edge crossing some cut of the graph.
Let T be a **minimum spanning tree** of graph G and let $(u, v) \in T$.

- Removing (u, v) from T splits it into **two connected components**, say A and B, such that:
  - $u \in A, v \in B$
  - The cut (A, B) separates the graph

Now:

- (u, v) is one of the edges **crossing this cut**
- Since T is a minimum spanning tree, **no other edge crossing the cut can have smaller weight**, or else replacing (u, v) would yield a lower-weight spanning tree

Therefore, (u, v) must be a **light edge** (minimum-weight edge) crossing the cut (A, B)

Give a simple example of a connected graph such that the set of edges $\{(u, v) :$ there exists a cut $(S, V - S)$ such that (u, v) is a light edge crossing $(S, V - S)\}$ does not form a minimum spanning tree.
This contradicts the **Cut Property** of Minimum Spanning Trees:

> If an edge (u, v) is the **lightest edge crossing any cut**, then it **must appear in some MST**.

So:

- If (u, v) is a **light edge for a cut**, it **must be included** in **some MST**
- Therefore, a graph with a light edge **not appearing in any MST cannot exist**

Let e be a maximum-weight edge on some cycle of connected graph $G = (V, E)$. Prove that there is a minimum spanning tree of $G' = (V, E - \{e\})$ that is also a minimum spanning tree of G. That is, there is a minimum spanning tree of G that does not include e.

1. Let C be a cycle in G, and $e \in C$ be the **maximum-weight edge**.
2. Suppose T is a minimum spanning tree that **does include** e.
3. Then there must be **another edge** $f \in C$ with weight **less than or equal to** $w(e)$.
4. Replace e with f in T, resulting in a new spanning tree T' with weight $\leq T$.
5. Since T was already an MST, T' must also be an MST.

Thus, we have constructed an MST **without edge** e.

Show that a graph has a unique minimum spanning tree if, for every cut of the graph, there is a unique light edge crossing the cut. Show that the converse is not true by giving a counterexample.

Claim: A graph **can have a unique MST** even if **some cuts** have **multiple light edges**

Counterexample: Graph with vertices: A, B, C

Edges:

- $A - B = 1$
- $B - C = 1$
- $A - C = 2$

Only one MST exists: edges $A - B$ and $B - C$

But consider cut $\{A\}, \{B, C\}$
→ Edges crossing: $A - B = 1$, $A - C = 2$
→ $A - B$ is **light**, but **not unique** in all cuts (e.g., for cut $\{C\}, \{A, B\}$, both $B - C$ and $A - C$ could be light)

Argue that if all edge weights of a graph are positive, then any subset of edges that connects all vertices and has minimum total weight must be a tree. Give an example to show that the same conclusion does not follow if we allow some weights to be nonpositive.

If all edge weights are **positive**, then:

- Any **cycle** increases total weight unnecessarily.
- Removing any edge from a cycle keeps the graph **connected** and **reduces** total weight.
- So, the minimal subset of edges that connects all vertices and minimizes weight must be:
  - **Acyclic**
  - **Spanning**

Therefore, it must be a **Minimum Spanning Tree (MST)**.

Let the graph have vertices A, B, C, and edges:

- $A - B = 1$
- $B - C = 1$
- $A - C = -2$

Subset of edges: $\{A - B, B - C, A - C\}$

- Forms a **cycle**
- Total weight: $1 + 1 + (-2) = 0$

But removing any edge makes weight **> 0**

So, the **minimum-weight subset** is **not a tree** if **nonpositive weights are allowed**

Let T be a minimum spanning tree of a graph G, and let L be the sorted list of the edge weights of T. Show that for any other minimum spanning tree T' of G, the list L is also the sorted list of edge weights of T'.

1. Let T and T' be two MSTs of G.
2. Suppose their edge weight lists (when sorted) differ.
3. Then, there exists an index i such that the first $i - 1$ edge weights agree, but the i-th weight in T is **less than** the i-th in T'.
4. Swap the heavier edge in T' with the lighter one in T. This gives a new tree with **lower total weight** than T' — contradicts minimality of T'.

So, no such difference can exist — all MSTs must have **identical multiset of edge weights** (and thus same sorted list).

Fibonacci heap is asymptotically faster.

Fibonacci heap is faster **iff**:

$$E \log V > E + V \log V \Rightarrow \log V > \frac{V \log V}{E} \Rightarrow \frac{E}{V} > 1$$

So, **when the average degree is greater than 1**, Fibonacci heap starts to outperform.

For a sparse graph $G = (V, E)$, where $|E| = \Theta(V)$, is the implementation of Prim's algorithm with a Fibonacci heap asymptotically faster than the binary-heap implementation? What about for a dense graph, where $|E| = \Theta(V^2)$? How must the sizes $|E|$ and $|V|$ be related for the Fibonacci-heap implementation to be asymptotically faster than the binary-heap implementation?

Binary Heap	$O(E \log V)$
Fibonacci Heap	$O(E + V \log V)$

Consider a simple implementation of Prim's algorithm that runs in $O(V^2)$ time when the graph $G = (V, E)$ is represented as an adjacency matrix.

```
PRIM(G)
1  Initialize key[v] ← ∞ for all vertices v
2  Initialize mstSet[v] ← FALSE for all vertices v
3  key[0] ← 0    // Start from vertex 0
4  for count = 1 to |V|:
5      u ← vertex not in mstSet with minimum key[u]
6      mstSet[u] ← TRUE
7      for each vertex v:
8          if G[u][v] ≠ 0 and mstSet[v] = FALSE and G[u][v] < key[v]:
9              key[v] ← G[u][v]
```

Suppose that a graph G has a minimum spanning tree already computed. How quickly can you update the minimum spanning tree upon adding a new vertex and incident edges to G?

T be the current MST of G

v be the **new vertex** added

k be the number of **edges from** v to existing vertices

To maintain the MST:

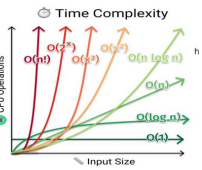
1. Find the **minimum-weight edge** (v, u) connecting v to any vertex in T
2. Add this edge to T

This is correct because:

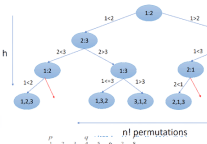
- T already spans all original vertices
- Adding v via the lightest edge preserves minimality and acyclicity

Optimal update time: $\boxed{O(k)}$

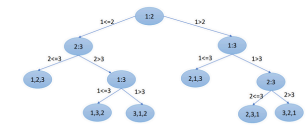
ALGORITHMS



Decision tree (Bubble Sort)



Decision tree (insertion Sort)



```
BRUTE-FORCE-FIND-MAXIMUM-SUBARRAY(A)
    n = A.length
    max-sum = -∞
    for l = 1 to n                    O(n^2)
        sum = 0
        for h = l to n
            sum = sum + A[h]
            if sum > max-sum
                max-sum = sum
                low = l
                high = h
    return (low, high, max-sum)
```

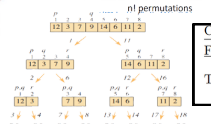
Algorithm 1 Divide-Conquer-Combine Algorithm

```
1:  function FindMaxSubarray(A, low, high)
2:      if low = high then
3:          return (low, high, A[low])
4:      else
5:          mid ← ⌊(low+high)/2⌋
6:          L ← FindMaxSubarray(A, low, mid)
7:          R ← FindMaxSubarray(A, mid + 1, high)
8:          C ← FMCS(A, low, mid, high)
9:          if L.maxSum >= R.maxSum and
10:             L.maxSum >= C.maxSum then
11:             return L
12:         else if R.maxSum >= L.maxSum and
13:             R.maxSum >= C.maxSum then
14:             return R
15:         else
16:             return C
17:         end if
18:     end if
19: end function
```

Algorithm 2 Find Maximum Crossing Subarray

```
1: function FMCS(A, low, mid, high)
2:     leftSum ← −∞
3:     sum ← 0
4:     for i ← mid downto low do
5:         sum ← sum + A[i]
6:         if sum > leftSum then
7:             leftSum ← sum
8:             maxLeft ← i
9:         end if
10:    end for
11:    rightSum ← −∞
12:    sum ← 0
13:    for j ← mid + 1 to high do
14:        sum ← sum + A[j]
15:        if sum > rightSum then
16:            rightSum ← sum
17:            maxRight ← j
18:        end if
19:    end for
20:    maxSum ← leftSum + rightSum
21:    return (maxLeft, maxRight,
       maxSum)
22: end function
```

```
MERGE-SORT(A, p, r)
1  if p ≥ r
2      return                          O(nlgn)
3  q = ⌊(p + r)/2⌋
4  MERGE-SORT(A, p, q)
5  MERGE-SORT(A, q + 1, r)
6  // Merge A[p : q] and A[q + 1 : r] i
7  MERGE(A, p, q, r)
```



Capacity constraint: For all $u, v \in V$, $0 \le f(u,v) \le c(u,v)$.

Flow conservation: For all $u \in V - \{s,t\}$, $\sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v)$.

The *value* $|f|$ of a flow f is defined as $|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s)$.

```
INSERTION-SORT(A, n)
for i = 2 to n                        O(n^2)
    key = A[i]
    // Insert A[i] into the sorted s
    j = i − 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j − 1
    A[j + 1] = key
```

```
MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)
1  if n == 1
2  // Base case.
3      c11 = c11 + a11 · b11
4      return
5  // Divide.
6  partition A, B, and C into n/2 × n/2 submatrices
       A11, A12, A21, A22; B11, B12, B21, B22;
       and C11, C12, C21, C22, respectively
```

```
TOPOLOGICAL-SORT(G)                   O(V+E)
1  call DFS(G) to compute finish times v.f for each vertex v
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2      print "A_i"
3  else print "("
4      PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5      PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6      print ")"
```

```
STRONGLY-CONNECTED-COMPONENTS(G)
1  call DFS(G) to compute finish times u.f for each vertex u
2  create G^T
3  call DFS(G^T), but in the main loop of DFS, consider the vertices
       in order of decreasing u.f (as computed in line 1)
4  output the vertices of each tree in the depth-first forest formed in line 3 as a
       separate strongly connected component
```

```
DFS-VISIT(G, u)                       O(V + E)
1  time = time + 1
2  u.d = time
3  u.color = GRAY
4  for each vertex v in G.Adj[u]
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  time = time + 1
9  u.f = time
10 u.color = BLACK
```

```
BFS(G, s)
1  for each vertex u ∈ G.V − {s}
2      u.color = WHITE
3      u.d = ∞
4      u.π = NIL
5  s.color = GRAY                     O(V + E)
6  s.d = 0
7  s.π = NIL
8  Q = ∅
9  ENQUEUE(Q, s)
10 while Q ≠ ∅                        O(nlgn)
11     u = DEQUEUE(Q)
12     for each vertex v in G.A
13         if v.color == WHITE
14             v.color = GRA
15             v.d = u.d + 1
16             v.π = u
17             ENQUEUE(Q, v
18     u.color = BLACK
```

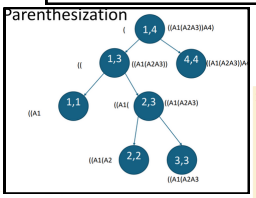
```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE                        O(VE)
```

```
BFS_main(G)
1 for each vertex u∈G.v
2     if u unexplored
3         BFS(G,u)
```

```
FORD-FULKERSON-METHOD(G, s, t)
1  initialize flow f to 0
2  while there exists an augmenting path p in the residual network G_f
3      augment flow f along p
4  return f
```

```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ G.E
2      (u, v).f = 0
3  while there exists a path p from s to t in the residual network G_f
4      c_f(p) = min{c_f(u, v) : (u, v) is in p}
5      for each edge (u, v) in p
6          if (u, v) ∈ G.E
7              (u, v).f = (u, v).f + c_f(p)
8          else (u, v).f = (v, u).f − c_f(p)
9  return f
```

Parenthesization




```
MATRIX-CHAIN-ORDER(p, n)
1  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
2  for i = 1 to n                     // chain length 1
3      m[i, i] = 0
4  for l = 2 to n                     // l is the chain length
5      for i = 1 to n − l + 1         // chain begins at A_i
6          j = i + l − 1              // O(n^3) ds at A_j
7          m[i, j] = ∞
8          for k = i to j − 1         // try A_{i:k} A_{k+1:j}
9              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10             if q < m[i, j]
11                 m[i, j] = q
12                 s[i, j] = k
13 return m and s
```

```
LCS-LENGTH(X, Y, m, n)
1  let b[1 : m, 1 : n] and c[1 : m, 1 : n] be new tables
2  for i = 1 to m
3      c[i, 0] = 0
4  for j = 0 to n
5      c[0, j] = 0
6  for i = 1 to m
7      for j = 1 to n
8          if x_i == y_j
9              c[i, j] = c[i − 1, j − 1] + 1
10             b[i, j] = "↖"
11         elseif c[i − 1, j] ≥ c[i, j − 1]
12             c[i, j] = c[i − 1, j]
13             b[i, j] = "↑"
14         else c[i, j] = c[i, j − 1]
15             b[i, j] = "←"
16 return c and b
```

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

```
MST-PRIM(G, w, r)
1  for each vertex u ∈ G.V
2      u.key = ∞
3      u.π = NIL
4  r.key = 0
5  Q = ∅
6  for each vertex u ∈ G.V
7      INSERT(Q, u)
8  while Q ≠ ∅
9      u = EXTRACT-MIN(Q)             // add u to
10     for each vertex v in G.Adj[u]  // update k
11         if v ∈ Q and w(u, v) < v.key
12             v.π = u
13             v.key = w(u, v)
14             DECREASE-KEY(Q, v, w(u, v))
```

$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E, \\ f(v,u) & \text{if } (v,u) \in E, \\ 0 & \text{otherwise.} \end{cases}$

$\forall u, v \in V$, We define the residual capacity $c_f(u,v)$ by

```
function tabulatedFib(n){
  if (n === 1 || n === 2){
    return 1;
  }
  const fibNums = [0, 1, 1];
  for (let i = 3; i <= n; i++){
    fibNums[i] = fibNums[i-1] + fibNums[i-2];
  }
  return fibNums[n];
}
```

Algorithm to Check if the Given Graph is a Bipartite Graph or Not

- Here is an algorithm for checking whether the given graph is a Bipartite or not using the BFS approach.
- **Step 1:** Start by picking an arbitrary vertex as the starting point and mark it as visited and add it to a queue.
- **Step 2:** Assign a color to the starting vertex (let's say color red).
- **Step 3:** While the queue is not empty, take the first vertex from the queue and look at its neighbors.
- **Step 4:** For each neighbor of the current vertex, check if it has already been visited or not.
- **Step 5:** If the neighbor has not been visited, mark it as visited, add it to the queue, and assign it the opposite color of its parent vertex. So, if the parent vertex is colored red, the neighbor will be colored blue, and vice versa.
- **Step 6:** If the neighbor has been visited and has the same color as its parent vertex, then the graph is not bipartite.
- **Step 7:** Repeat steps 3 to 6 until all vertices have been visited or until a non-bipartite condition is detected.
- **Step 8:** If all vertices have been visited without any non-bipartite condition, then the graph is bipartite.

```
HUFFMAN(C)
1  n = |C|                            O(nlgn)
2  Q = C
3  for i = 1 to n − 1
4      allocate a new node z
5      z.left = x = EXTRACT-MIN(Q)
6      z.right = y = EXTRACT-MIN(Q)
7      z.freq = x.freq + y.freq
8      INSERT(Q, z)
9  return EXTRACT-MIN(Q)    // return the root of the tree
```

$$T(n) = \sum_{k=0}^{\log_b n - 1} a^k \cdot f\left(\frac{n}{b^k}\right) + \Theta(n^{\log_b a})$$

```
MATRIX-MULTIPLY(A, B, C, n)
1  for i = 1 to n                     //
2      for j = 1 to n                 O(n^3) //
3          for k = 1 to n
4              c_ij = c_ij + a_ik · b_kj //
```

Case 1: f(n)< $n^{lg_b a}$

- T(n)= $\sum_{k=0}^{lg_b n-1} a^k f\left(\frac{n}{b^k}\right) + \Theta(n^{lg_b a})$ → eq1
- By Case 1:
  - If f(n)= $O(n^{lg_b a - \epsilon})$ implies $f\left(\frac{n}{b^k}\right) = O\left(\left(\frac{n}{b^k}\right)^{lg_b a - \epsilon}\right)$ substitute in eq1
- T(n)<= $\sum_{k=0}^{lg_b n-1} a^k \left(\frac{n}{b^k}\right)^{lg_b a - \epsilon} + \Theta(n^{lg_b a})$
- T(n)<= $n^{lg_b a - \epsilon} \sum_{k=0}^{lg_b n-1} a^k \left(\frac{1}{b^k}\right)^{lg_b a - \epsilon} + \Theta(n^{lg_b a})$
- T(n)<= $n^{lg_b a - \epsilon} \sum_{k=0}^{lg_b n-1} (a \left(\frac{b^\epsilon}{b^{lg_b a}}\right)^k + \Theta(n^{lg_b a})$ [by $b^{lg_b a} = a$]
- T(n)<= $n^{lg_b a - \epsilon} \sum_{k=0}^{lg_b n-1} (a \left(\frac{b^\epsilon}{a}\right)^k + \Theta(n^{lg_b a})$ [by $b^{lg_b a} = a$]
- T(n)<= $n^{lg_b a - \epsilon} \sum_{k=0}^{lg_b n-1} (b^\epsilon)^k + \Theta(n^{lg_b a})$
- T(n)<= $n^{lg_b a - \epsilon} \frac{b^{\epsilon lg_b n} - 1}{b^\epsilon - 1} + \Theta(n^{lg_b a})$ [by geometric series]
- T(n)<= $n^{lg_b a - \epsilon} \frac{b^{\epsilon lg_b n} - 1}{b^\epsilon - 1} + \Theta(n^{lg_b a})$
- T(n)<= $n^{lg_b a - \epsilon} \frac{n^\epsilon - 1}{b^\epsilon - 1} + \Theta(n^{lg_b a})$[by disregarding contant terms b and ϵ]
- T(n)= $\Theta(n^{lg_b a})$

$\sum_{k=1}^{n} x^k = \frac{x^n - 1}{x - 1}$

Case 2: f(n)<= $n^{lg_b a}$

- T(n)= $\sum_{k=0}^{lg_b n-1} a^k f\left(\frac{n}{b^k}\right) + \Theta(n^{lg_b a})$ [substitute $f\left(\frac{n}{b^k}\right) = \Theta\left(\frac{n}{b^k}\right)^{lg_b a}$]
- = $n^{lg_b a} \sum_{k=0}^{lg_b n-1} a^k \left(\frac{1}{b^k}\right)^{lg_b a} + \Theta(n^{lg_b a})$
- =$n^{lg_b a} \sum_{k=0}^{lg_b n-1} a^k (b^{-k})^{lg_b a} + \Theta(n^{lg_b a})$ [by $b^{lg_b a} = a$]
- = $n^{lg_b a} \sum_{k=0}^{lg_b n-1} a^k (a^{-k}) + \Theta(n^{lg_b a})$
- =$n^{lg_b a} (lg_b n + 1) + \Theta(n^{lg_b a})$ [take n-1≈n]
- = $\Theta(n^{lg_b a} \, lg_b n)$

Case 3: f(n)> $n^{lg_b a}$

- f(n)=Ω ($n^{lg_b a + \epsilon}$)for any ϵ>0 with c= $b^{-\epsilon}$<1 and af($\frac{n}{b}$)<=cf(n)
- af($\frac{n}{b}$)=a$(\frac{n}{b})^{lg_b a + \epsilon}$=a$n^{lg_b a + \epsilon} b^{-lg_b a} b^{-\epsilon} = a \frac{n^{lg_b a + \epsilon}}{b^{lg_b a}} b^{-\epsilon}$
- = $n^{lg_b a + \epsilon} b^{-\epsilon}$ =f(n) $b^{-\epsilon}$
- a^kf($\frac{n}{b^k}$)<=c^kf(n) [af($\frac{n}{b}$)<=cf(n)]
- T(n)= $\sum_{k=0}^{lg_b n-1} a^k f\left(\frac{n}{b^k}\right) + \Theta(n^{lg_b a})$ → eq1
- T(n)<= $\sum_{k=0}^{lg_b n-1} c^k$f(n) +$\Theta(n^{lg_b a})$
- T(n)<= f(n)$\sum_{k=0}^{\infty} c^k$ +$\Theta(n^{lg_b a})$
- = $\Theta(n^{lg_b a})$+ f(n)$\frac{1}{1-c}$ [Sum of infinite geometric series a+ar+ar^2+ar^3... = a / (1 - r)
- = Θ(f(n))

Coin Change (min coins)	$dp[i] = \min(dp[i], dp[i - coin] + 1)$ fo
Palindrome Partitioning	$dp[i] = \min_{j<i}(dp[j] + 1)$ if $s[j+1..i]$

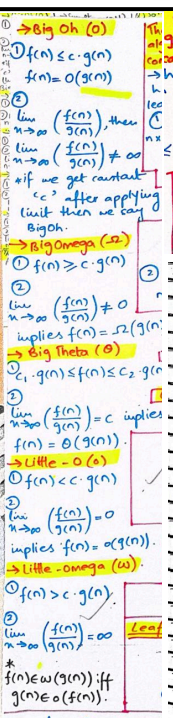
Master Theorem

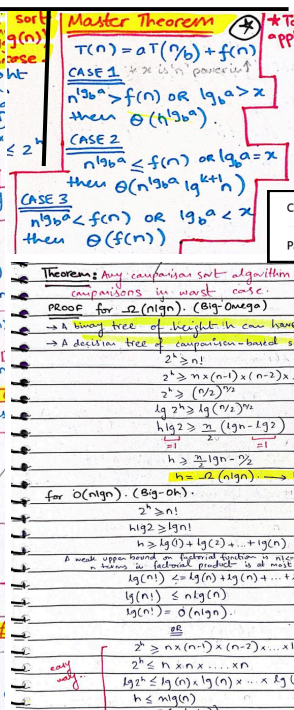
$T(n) = aT(n/b) + f(n)$

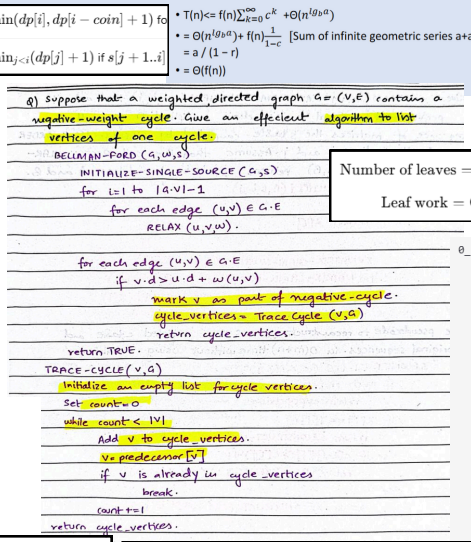
CASE 1: $n^{lg_b a} > f(n)$ OR $lg_b a > x$ then $\Theta(n^{lg_b a})$.

CASE 2: $n^{lg_b a} \le f(n)$ OR $lg_b a = x$ then $\Theta(n^{lg_b a} lg^{k+1} n)$

CASE 3: $n^{lg_b a} < f(n)$ OR $lg_b a < x$ then $\Theta(f(n))$







Number of leaves = $a^{\log_b n} = n^{\log_b a}$

Leaf work = $\Theta(n^{\log_b a})$

```
FRACTIONAL-KNAPSACK(items, W):
    # Each item has (value, weight)
    for each item:                    O(nlgn)
        compute value_per_weight = value / weight

    sort items in decreasing order of value_per_weight

    total_value = 0
    for item in sorted items:
        if W == 0:
            break

        if item.weight <= W:
            take full item
            total_value += item.value
            W -= item.weight
        else:
            take fraction of item
            total_value += item.value_per_weight * W
            W = 0

    return total_value
```

```
0_1_KNAPSACK(values, weights, n, W):
    Create a 2D array dp[n+1][W+1]

    for i from 0 to n:                O(2^n)
        for w from 0 to W:
            if i == 0 or w == 0:
                dp[i][w] = 0  # Base case: zero items or zero capacity
            else if weights[i-1] <= w:
                dp[i][w] = max(
                    values[i-1] + dp[i-1][w - weights[i-1]],  # take item
                    dp[i-1][w]                                # skip item
                )
            else:
                dp[i][w] = dp[i-1][w]  # can't take item, too heavy

    return dp[n][W]  # Maximum value we can get
```

```
EDMONDS-KARP(G, s, t):
    Initialize flow f(u, v) = 0 for all edges (u, v) in G
    Initialize max_flow = 0

    while there exists an augmenting path P from s to t (found using BFS in
        # Find minimum residual capacity along the path
        path_flow = ∞
        for each edge (u, v) in path P:
            path_flow = min(path_flow, residual_capacity(u, v))

        # Augment the flow along the path
        for each edge (u, v) in path P:
            f(u, v) += path_flow
            f(v, u) -= path_flow  # Update reverse flow for residual graph

        max_flow += path_flow

    return max_flow
```

22.1-4

Modify the Bellman-Ford algorithm so that it sets $v.d$ to $-\infty$ for all vertices v fo which there is a negative-weight cycle on some path from the source to v.

After the standard $|V| - 1$ passes, add a **final pass** to detect and propagate $-\infty$:

```
for each edge (u, v) ∈ E:
    if d[v] > d[u] + w(u, v):
        d[v] ← −∞
```

Then, **propagate $-\infty$ to all reachable nodes** using BFS/DFS:

```
for each vertex v with d[u] = −∞:
    for each vertex v reachable from u:
        d[v] ← −∞
```

Definition 5 (Flow Augmentation). If f is a flow in G and f' is a flow in the corresponding residual network G_f, we define $f \uparrow f'$, the *augmentation* of flow f by f', to be a function from $V \times V$ to $\mathbb{R}$, defined by

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{if } (u,v) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Lemma 24.2. *Let $G = (V, E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f. Define a function $f_p : V \times V \to \mathbb{R}$ by*

$$f_p(u,v) = \begin{cases} c_f(p) & \text{if } (u,v) \text{ is on } p, \\ 0 & \text{otherwise.} \end{cases}$$

Then, f_p is a flow in G_f with value $|f_p| = c_f(p) > 0$.

Definition 6 (Cut of a Flow Network). A *cut* (S, T) of flow network $G = (V, E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$. If f is a flow, then the *net flow* $f(S, T)$ across the cut (S, T) is defined to be

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u).$$

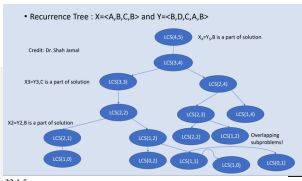
The *capacity* of the cut (S, T) is

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v).$$

A minimum cut of a network is a cut whose capacity is minimum over all cuts of the network.

Strasser's Algorithm

1. If $n = 1$, the matrices each contain a single element. Perform a single scalar multiplication and a single scalar addition, as in line 3 of MATRIX-MULTIPLY-RECURSIVE, taking $\Theta(1)$ time, and each edge indicates the vertices it leaves and enters, along with its weight. Otherwise, partition the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (4.2). This step takes $\Theta(1)$ time by index calculation, just as in MATRIX-MULTIPLY-RECURSIVE.
2. Create $n/2 \times n/2$ matrices S_1 S_2 . . . S_{10}, each of which is the sum or difference of two submatrices from step 1. Create and zero the entries of seven $n/2 \times n/2$ matrices P_1 P_2 . . . P_7 to hold seven $n/2 \times n/2$ matrix products. All 17 matrices can be created, and the P_i initialized, in $\Theta(n^2)$ time.
3. Using the submatrices from step 1 and the matrices $S_1, S_2, \ldots, S_{10}$ created in step 2, recursively compute each of the seven matrix products $P_1, P_2, \ldots, P_7$, taking $7T(n/2)$ time.



22.1-5

Suppose that the graph given as input to the Bellman-Ford algorithm is represented with a list of $|E|$ edges, where each edge indicates the vertices it leaves and enters, along with its weight. Argue that the Bellman-Ford algorithm runs in $O(VE)$ time without the constraint that $|E| = \Omega(V)$. Modify the Bellman-Ford algorithm so that it runs in $O(VE)$ time in all cases when the input graph is represented with adjacency lists.

☑ **Claim:**

Bellman-Ford runs in $O(VE)$ even without $|E| = \Omega(V)$,

because it performs $V - 1$ passes over all E edges.

Instead of iterating over an edge list:

```
for u in V:
    for each v in Adj[u]:
        relax(u, v)
```

☑ This ensures each pass takes $O(E)$ time.

☑ Total time:

$O(VE)$

22.1-3

Given a weighted, directed graph $G = (V, E)$ with no negative-weight cycles, let m be the maximum over all vertices $v \in V$ of the minimum number of edges in a shortest path from the source s to v. (Here, the shortest path is by weight, not the number of edges.) Suggest a simple change to the Bellman-Ford algorithm that allows it to terminate in $m + 1$ passes, even if m is not known in advance.

Add a boolean flag `updated` to track whether any distance was changed during a pass.

Terminate the algorithm **when no updates occur** in a full pass.

```
for i = 1 to ∞:
    updated ← FALSE
    for each edge (u, v) ∈ E:
        if d[v] > d[u] + w(u, v):
            d[v] ← d[u] + w(u, v)
            updated ← TRUE
    if not updated:
        break
```

Edit Distance	$dp[i][j] = \min \begin{cases} dp[i-1][j] + 1 \\ dp[i][j-1] + 1 \\ dp[i-1][j-1] + (X[i-1] \neq Y[j-1]) \end{cases}$
Longest Common Subsequence (LCS)	$dp[i][j] = dp[i-1][j-1] + 1$ if $X[i] = Y[j]$, else $\max(dp[i-1][j], dp[i][j-1])$
Longest Common Substring	$dp[i][j] = dp[i-1][j-1] + 1$ if $X[i] = Y[j]$, else 0
Matrix Chain Multiplication	$dp[i][j] = \min_{i \le k < j}(dp[i][k] + dp[k+1][j] + p_{i-1} p_k p_j)$
0/1 Knapsack	$dp[i][w] = \max(dp[i-1][w], val_i + dp[i-1][w - wt_i])$
Subset Sum	$dp[i][s] = dp[i-1][s] \vee dp[i-1][s - A[i]]$
Rod Cutting	$dp[n] = \max_{1 \le i \le n}(price[i] + dp[n - i])$
Longest Increasing Subsequence (LIS)	$L(i) = 1 + \max(L(j))$ for all $j < i$ and $A[j] < A[i]$
Longest Increasing Contiguous Subsequence (LICS)	$L(i) = L(i-1) + 1$ if $A[i] > A[i-1]$, else 1