1. Amortized Analysis of a k-bit Binary Counter (Aggregate Method)

Problem Definition: Consider a k-bit binary counter that counts upward from 0. We start with an integer represented by an array $A[0 \dots k-1]$ of bits, initialized to 0. We perform a sequence of n `INCREMENT` operations. The cost of an `INCREMENT` operation is the number of bits flipped.

- Changing $0 \to 1$ costs 1 unit.
- Changing $1 \to 0$ costs 1 unit.

Naive Analysis (Worst Case): In the worst case, an `INCREMENT` operation might flip all k bits (e.g., incrementing $11 \dots 1$ to $10 \dots 0$). Therefore, for n operations, the worst-case cost is $O(n \cdot k)$. However, this is too pessimistic because not every operation flips k bits.

Aggregate Analysis (Step-by-Step): The aggregate method calculates the total cost of a sequence of n operations and divides it by n to find the amortized cost.

1. **Bit Flipping Pattern:**
   - **Bit 0 ($A[0]$):** Flips every time ($0 \to 1, 1 \to 2$, etc.). In n operations, it flips n times.
   - **Bit 1 ($A[1]$):** Flips every 2^{nd} time ($1 \to 2, 3 \to 4$). In n operations, it flips $\lfloor n/2 \rfloor$ times.
   - **Bit 2 ($A[2]$):** Flips every 4^{th} time ($3 \to 4, 7 \to 8$). In n operations, it flips $\lfloor n/4 \rfloor$ times.
   - ...
   - **Bit i ($A[i]$):** Flips every 2^i-th time. In n operations, it flips $\lfloor n/2^i \rfloor$ times.
2. **Total Cost Calculation:** The total number of flips for n operations is the sum of flips for each bit position i from 0 to $k-1$:

$$\text{Total Cost} = \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor$$

   Since $\lfloor n/2^i \rfloor \le n/2^i$, we can write:

$$\text{Total Cost} < \sum_{i=0}^{\infty} \frac{n}{2^i}$$

$$\text{Total Cost} < n \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i$$

   Using the geometric series formula $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ for $x = 1/2$:

$$\text{Total Cost} < n \left(\frac{1}{1-1/2}\right) = n(2) = 2n$$

3. **Amortized Cost:**

$$\text{Amortized Cost} = \frac{\text{Total Cost}}{n} \le \frac{2n}{n} = 2$$

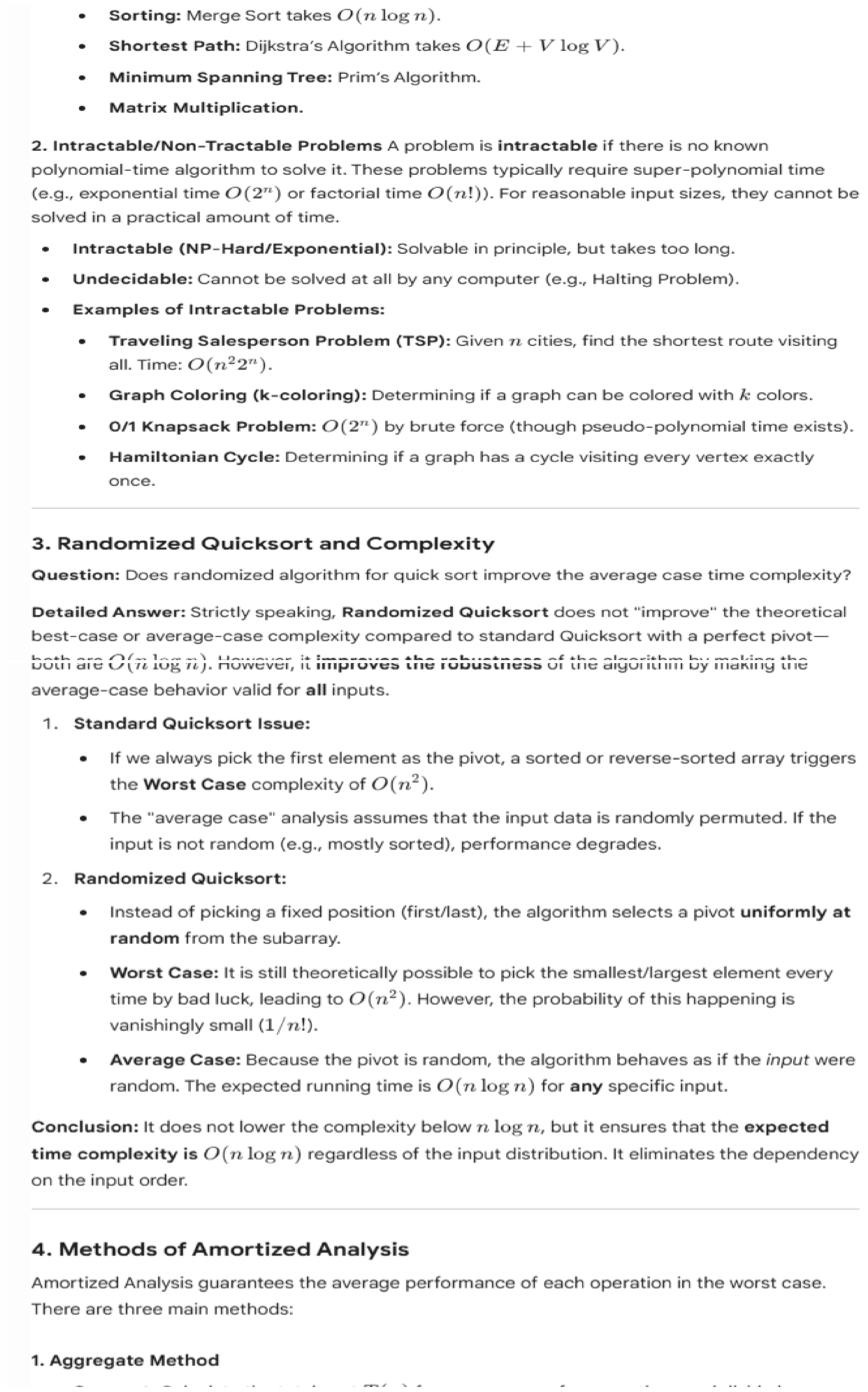
Conclusion: Using the aggregate method, we proved that while a single operation can cost k, the average (amortized) cost over any sequence of operations is constant, specifically $O(1)$.

---

2. Tractable and Non-Tractable Problems

1. Tractable Problems (Class P) A problem is considered **tractable** if there exists a polynomial-time algorithm to solve it. This means the running time of the algorithm is bounded by a polynomial function of the input size n (e.g., $O(n)$, $O(n^2)$, $O(n^{100})$).

- **Characteristics:** Efficiently solvable for large inputs.
- **Examples:**



- **Sorting:** Merge Sort takes $O(n \log n)$.
- **Shortest Path:** Dijkstra's Algorithm takes $O(E + V \log V)$.
- **Minimum Spanning Tree:** Prim's Algorithm.
- **Matrix Multiplication.**

2. Intractable/Non-Tractable Problems A problem is **intractable** if there is no known polynomial-time algorithm to solve it. These problems typically require super-polynomial time (e.g., exponential time $O(2^n)$ or factorial time $O(n!)$). For reasonable input sizes, they cannot be solved in a practical amount of time.

- **Intractable (NP-Hard/Exponential):** Solvable in principle, but takes too long.
- **Undecidable:** Cannot be solved at all by any computer (e.g., Halting Problem).
- **Examples of Intractable Problems:**
  - **Traveling Salesperson Problem (TSP):** Given n cities, find the shortest route visiting all. Time: $O(n^2 2^n)$.
  - **Graph Coloring (k-coloring):** Determining if a graph can be colored with k colors.
  - **0/1 Knapsack Problem:** $O(2^n)$ by brute force (though pseudo-polynomial time exists).
  - **Hamiltonian Cycle:** Determining if a graph has a cycle visiting every vertex exactly once.

3. Randomized Quicksort and Complexity

Question: Does randomized algorithm for quick sort improve the average case time complexity?

Detailed Answer: Strictly speaking, **Randomized Quicksort** does not "improve" the theoretical best-case or average-case complexity compared to standard Quicksort with a perfect pivot—both are $O(n \log n)$. However, it **improves the robustness** of the algorithm by making the average-case behavior valid for **all** inputs.

1. **Standard Quicksort Issue:**
   - If we always pick the first element as the pivot, a sorted or reverse-sorted array triggers the **Worst Case** complexity of $O(n^2)$.
   - The "average case" analysis assumes that the input data is randomly permuted. If the input is not random (e.g., mostly sorted), performance degrades.
2. **Randomized Quicksort:**
   - Instead of picking a fixed position (first/last), the algorithm selects a pivot **uniformly at random** from the subarray.
   - **Worst Case:** It is still theoretically possible to pick the smallest/largest element every time by bad luck, leading to $O(n^2)$. However, the probability of this happening is vanishingly small ($1/n!$).
   - **Average Case:** Because the pivot is random, the algorithm behaves as if the *input* were random. The expected running time is $O(n \log n)$ for **any** specific input.

Conclusion: It does not lower the complexity below $n \log n$, but it ensures that the **expected time complexity is** $O(n \log n)$ regardless of the input distribution. It eliminates the dependency on the input order.

4. Methods of Amortized Analysis

Amortized Analysis guarantees the average performance of each operation in the worst case. There are three main methods:

1. Aggregate Method

- **Concept:** Calculate the total cost $T(n)$ for a sequence of n operations and divide by n.
- **Formula:** $\text{Amortized Cost} = T(n)/n$.
- **Example: Dynamic Array Expansion**.
  - Most insertions take $O(1)$. Occasionally, the array doubles (cost $O(n)$).
  - Total cost for n insertions is $3n$.
  - Amortized cost per insertion = $3n/n = 3 = O(1)$.

2. Accounting Method (Banker's Method)

- **Concept:** We assign different charges to different operations.
  - **Amortized Cost > Actual Cost:** The surplus is stored as "credit" on specific elements in the data structure.
  - **Amortized Cost < Actual Cost:** The stored credit is used to pay for the expensive operation.
- **Example: Stack Operations (Push, Pop, MultiPop)**.
  - Charge $2 for PUSH. (Actual cost $1, save $1 credit on the element).
  - Charge $0 for POP. (Actual cost $1, pay using the $1 credit stored on the element being popped).
  - We never run out of credit because we can't pop an element that wasn't pushed.

3. Potential Function Method (Physicist's Method)

- **Concept:** We define a potential function $\Phi(D)$ representing the "energy" or "state" of the data structure.
- **Formula:** $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
  - Where $\hat{c}_i$ is amortized cost, c_i is actual cost.
- **Example: Binary Counter**.
  - $\Phi(D) = $ number of 1s in the counter.
  - When we flip bits $1 \to 0$, potential decreases, paying for the actual work.
  - Resulting amortized cost is 2.

---

5. Approximation Algorithms & Performance Ratios

Definition: An **Approximation Algorithm** is a way of dealing with NP-Hard optimization problems. Since we cannot find the exact optimal solution in polynomial time, we settle for a solution that is "close enough" to the optimal, achievable in polynomial time.

Performance Ratios (Approximation Ratio): The performance ratio $\rho(n)$ measures how bad the approximate solution can be compared to the optimal solution.

Let C be the cost of the solution found by the approximation algorithm. Let C^* be the cost of the optimal solution.

- **For Minimization Problems:**

$$\frac{C}{C^*} \leq \rho(n)$$

  (We want C to be small, so ratio ≥ 1).

- **For Maximization Problems:**

$$\frac{C^*}{C} \leq \rho(n)$$

  (We want C to be large, close to C^*).

Utility of Performance Ratios:

1. **Guarantees Quality:** It provides a worst-case bound. If an algorithm is a 2-approximation for Vertex Cover, we know our solution is never more than twice the size of the best possible one.
2. **Classification:** It helps classify NP-Hard problems based on difficulty. Some allow constant ratios (APX), others only logarithmic (Set Cover), and some cannot be approximated well at all (TSP).

---

6. Randomized Algorithms

Definition: A **Randomized Algorithm** is an algorithm that employs a degree of randomness as part of its logic. It uses a random number generator to make decisions (e.g., picking a pivot in QuickSort, choosing a hash function).

Primary Reasons for Using Randomized Algorithms:

1. **Simplicity:** They are often much simpler to implement than their deterministic counterparts. (e.g., Randomized QuickSort is simpler than the deterministic "Median-of-Medians" algorithm for $O(n)$ selection).
2. **Efficiency:** They can be faster in practice. (e.g., Randomized Primality Testing is faster than deterministic methods).
3. **Symmetry Breaking:** In distributed systems, randomness helps break deadlocks (e.g., Ethernet backoff protocol).
4. **Adversary Proof:** By making random choices, the algorithm prevents an "adversary" from constructing a specific input that triggers worst-case behavior.

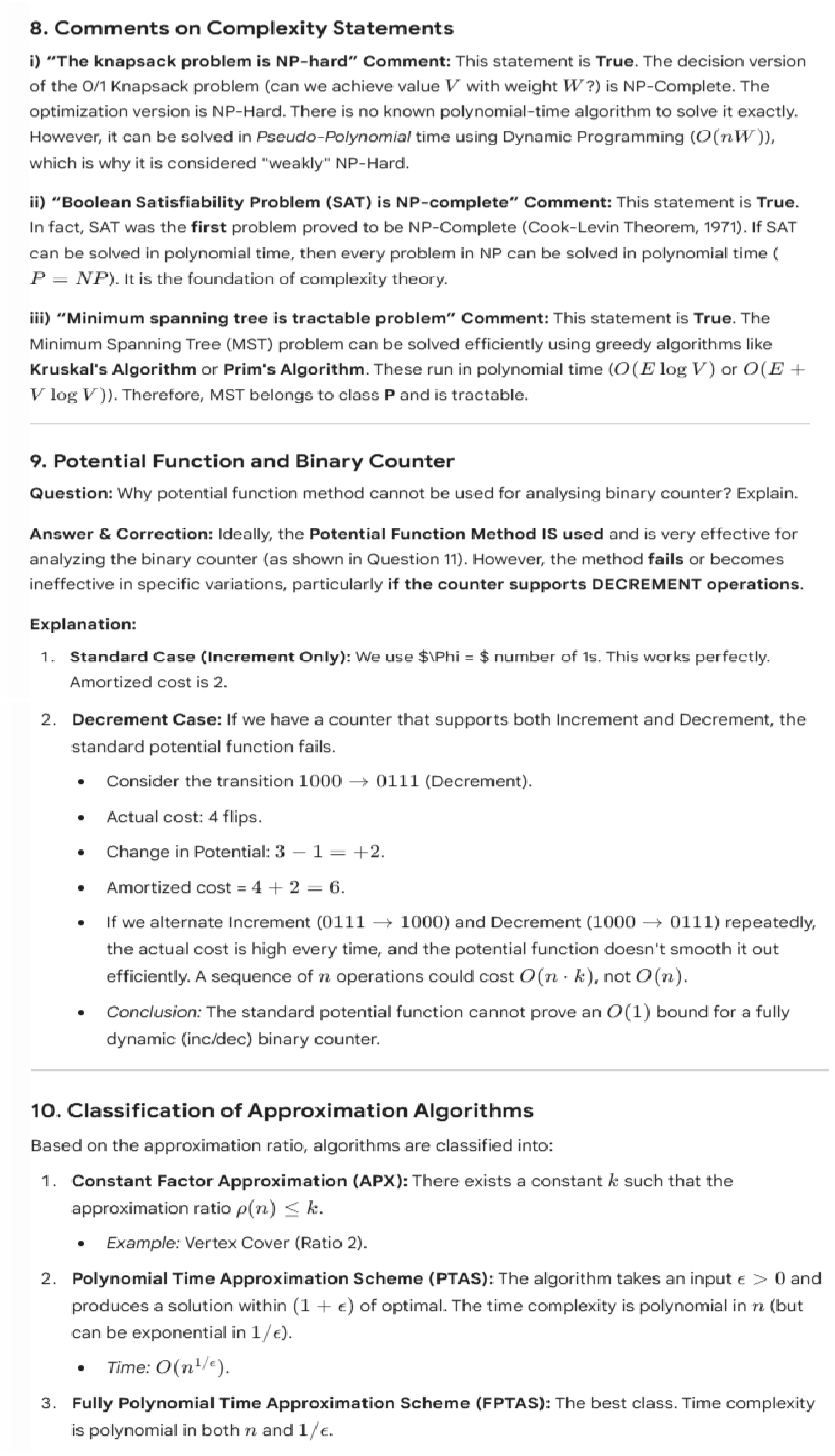
Types:

- **Las Vegas:** Always produces the correct result; time varies (e.g., QuickSort).
- **Monte Carlo:** Runs in fixed time; result might be wrong with small probability (e.g., Karger's Min-Cut).

---

7. Comparison: Aggregate vs. Accounting Method

Feature	i) Aggregate Analysis	ii) Accounting Method
Concept	Computes the average cost over a sequence of operations directly.	Assigns "charges" (amortized costs) to operations; saves surplus as credit.
Cost assignment	All operations get the same amortized cost ($T(n)/n$).	Different operations can have different amortized costs.
Complexity	Simple for problems where total cost is easy to sum.	Requires intuition to assign the correct charges/credits.
State Tracking	Does not track the state of individual elements.	Must track "credit" stored on specific elements/objects.
Precision	Gives a global average.	Can prove bounds for specific operation types.
Flexibility	Less flexible; purely analytical.	More flexible; resembles a "bank account" metaphor.
Advantage	Easiest to apply if the total sum series is known.	Good when specific operations (like POP) are clearly "free" due to earlier work.
Disadvantage	Does not explain *why* an operation is cheap/expensive.	If charges are set wrong, the proof fails (credits go negative).



8. Comments on Complexity Statements

i) "The knapsack problem is NP-hard" Comment: This statement is **True**. The decision version of the 0/1 Knapsack problem (can we achieve value V with weight W?) is NP-Complete. The optimization version is NP-Hard. There is no known polynomial-time algorithm to solve it exactly. However, it can be solved in *Pseudo-Polynomial* time using Dynamic Programming ($O(nW)$), which is why it is considered "weakly" NP-Hard.

ii) "Boolean Satisfiability Problem (SAT) is NP-complete" Comment: This statement is **True**. In fact, SAT was the **first** problem proved to be NP-Complete (Cook-Levin Theorem, 1971). If SAT can be solved in polynomial time, then every problem in NP can be solved in polynomial time ($P = NP$). It is the foundation of complexity theory.

iii) "Minimum spanning tree is tractable problem" Comment: This statement is **True**. The Minimum Spanning Tree (MST) problem can be solved efficiently using greedy algorithms like **Kruskal's Algorithm** or **Prim's Algorithm**. These run in polynomial time ($O(E \log V)$ or $O(E + V \log V)$). Therefore, MST belongs to class **P** and is tractable.

---

9. Potential Function and Binary Counter

Question: Why potential function method cannot be used for analysing binary counter? Explain.

Answer & Correction: Ideally, the **Potential Function Method IS used** and is very effective for analyzing the binary counter (as shown in Question 11). However, the method **fails** or becomes ineffective in specific variations, particularly **if the counter supports DECREMENT operations.**

Explanation:

1. **Standard Case (Increment Only):** We use $\Phi = $ number of 1s. This works perfectly. Amortized cost is 2.
2. **Decrement Case:** If we have a counter that supports both Increment and Decrement, the standard potential function fails.
   - Consider the transition $1000 \to 0111$ (Decrement).
   - Actual cost: 4 flips.
   - Change in Potential: $3 - 1 = +2$.
   - Amortized cost = $4 + 2 = 6$.
   - If we alternate Increment ($0111 \to 1000$) and Decrement ($1000 \to 0111$) repeatedly, the actual cost is high every time, and the potential function doesn't smooth it out efficiently. A sequence of n operations could cost $O(n \cdot k)$, not $O(n)$.
   - *Conclusion:* The standard potential function cannot prove an $O(1)$ bound for a fully dynamic (inc/dec) binary counter.

---

10. Classification of Approximation Algorithms

Based on the approximation ratio, algorithms are classified into:

1. **Constant Factor Approximation (APX):** There exists a constant k such that the approximation ratio $\rho(n) \leq k$.
   - *Example:* Vertex Cover (Ratio 2).
2. **Polynomial Time Approximation Scheme (PTAS):** The algorithm takes an input $\epsilon > 0$ and produces a solution within $(1 + \epsilon)$ of optimal. The time complexity is polynomial in n (but can be exponential in $1/\epsilon$).
   - *Time:* $O(n^{1/\epsilon})$.
3. **Fully Polynomial Time Approximation Scheme (FPTAS):** The best class. Time complexity is polynomial in both n and $1/\epsilon$.

- *Example:* 0/1 Knapsack Problem.

4. **Logarithmic Approximation:** The ratio $\rho(n)$ grows with $\log n$.
   - *Example:* Set Cover Problem ($\ln n$).

---

11. Potential Function Method (Stack Operations)

Definition: We define a potential function Φ that maps the data structure D to a real number $\Phi(D)$. Amortized Cost $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$. Condition: $\Phi(D_i) \geq \Phi(D_0)$ for all i.

Analysis of Stack: Define $\Phi(S) = $ number of items in the stack. $\Phi(S_0) = 0$.

1. **PUSH(S, x):**
   - Actual cost $c = 1$.
   - Change in potential: Stack size increases by 1. $\Phi(D_i) - \Phi(D_{i-1}) = 1$.
   - **Amortized cost:** $\hat{c} = 1 + 1 = 2$.
2. **POP(S):**
   - Actual cost $c = 1$.
   - Change in potential: Stack size decreases by 1. $\Phi(D_i) - \Phi(D_{i-1}) = -1$.
   - **Amortized cost:** $\hat{c} = 1 + (-1) = 0$.
3. **MULTIPOP(S, k):**
   - Actual cost $c = k$ (pops k items).
   - Change in potential: Stack size decreases by k. $\Phi(D_i) - \Phi(D_{i-1}) = -k$.
   - **Amortized cost:** $\hat{c} = k + (-k) = 0$.

Conclusion: All operations have $O(1)$ amortized cost.

---

12. Amortized Analysis & Aggregate Method

What is Amortized Analysis? Amortized analysis is a technique used to analyze algorithms that perform a sequence of operations. It calculates the average cost per operation over the worst-case sequence. Unlike average-case analysis (which relies on probability), amortized analysis guarantees the average performance even for worst-case inputs.

Aggregate Method Explanation: The Aggregate Method is the simplest form.

1. We look at the entire sequence of n operations.
2. We calculate the worst-case **Total Cost** $T(n)$ for these n operations.
3. The amortized cost is simply the average: $T(n)/n$. This cost applies to every operation in the sequence, regardless of whether it was actually cheap or expensive.

Example: Stack with MultiPop

- Operations: PUSH (Cost 1), POP (Cost 1), MULTIPOP(k) (Cost k).
- Sequence: n operations total.
- **Observation:** We can only pop an item if it has been pushed. If we push n items total, we can pop at most n items total (whether via single POP or MULTIPOP).
- **Total Cost:** Total Pushes (n) + Total Pops ($\leq n$) $\leq 2n$.
- **Amortized Cost:** $2n/n = 2 = O(1)$.
- Even though `MULTIPOP` can cost $O(n)$ in the worst case, it cannot happen often enough to drag the average down.