Algorithms, Design & Analysis

Lecture 24: **DP Problems and NP Hardness**

Dr Mudassir Shabbir

Information Technology University

April 25, 2025

About Your Fellows

- Hi there! We are **Mihab Khan** and **Khadijah Farooqi**.
- We are Associate Students at ITU.

Quote by Edsger Dijkstra

"Computer Science is no more about computers than astronomy is about telescopes."

— Edsger W. Dijkstra

Side Talk

In fact, one of the greatest computer scientists, he didn't know how to check his emails. His student used to check his emails for him and respond to them. He used to make his slides on paper, and they would be put on a special projector for display to the people. And he is most probably the most influential computer scientist alive.

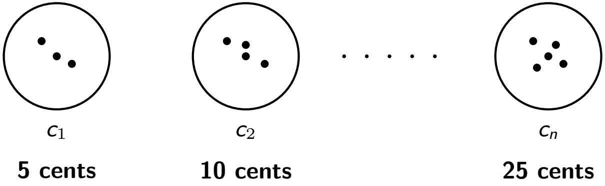
Topic 1:

Coin Change Problem

Coin Change Problem

Problem Statement

We have n piles of coins, each with an infinite supply. The denominations of the coins are given. A person demands A cents. The goal is to determine the minimum number of coins required to make A cents.



Infinite coins in each

- Assuming the coins are sorted.

It's possible that no combination adds up to A. For example, if $A = 32$ and we cannot make it using the given denominations, the algorithm returns ∞.

Greedy Approach

Idea

At each step, pick the largest coin value that is $\leq A$ and subtract it from A.

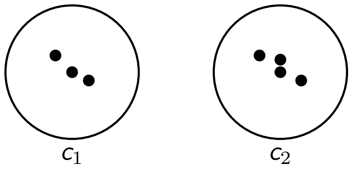
Algorithm

- Let the coin denominations be $C = [c_1, c_2, \ldots, c_n]$
- Sort the coin denominations such that $c_1 < c_2 < \cdots < c_n$
- Find largest i such that: $Ci < A$
- Then: $A' = A - k \cdot Ci$, where $A' < Ci$
- Recursively solve on A'
- Time Complexity: $\Theta(n \log n)$

Greedy Approach

Why doesn't this always work?
Hamza Naveed assumed it was impossible — but smaller coins could have solved it.

A = 9



3 cents **8 cents**

Greedy picks 8 cents first and gets stuck at 1
but choosing 3 cents three times (3 + 3 + 3) would have worked!

Dynamic Program

Basic Idea: Basically, in DP, we explore all possibilities, either recursively or using recurrence relations.

Recursive Formulation:
`DP([C1, C2, ..., CN], A)`
$\forall i$: return $\min(1 + \text{DP}([C1, C2, ..., CN], A - C_i))$

Basit assured:

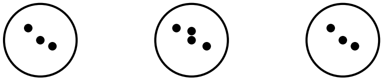
- `if A == 0, return 0`
- `if A < 0, return infinity`

Hamza Naveed raised a question — he was unsure about the following statement:

```
for (i = 1 to n):
    return min(1 + DP([C1, C2, ..., CN], A - C))
```

Hamza's confusion was about why we are exploring all the coins. How is it working? So, **Professor decided to explain it with an example.**

Example



c_1 **3 cents** | c_2 **7 cents** | c_3 **15 cents** | **A = 32**

Steps: 1) $1 + \text{DP}(29) \rightarrow 26 \rightarrow 23 \rightarrow 20 \rightarrow 17 \rightarrow 14 \rightarrow 11 \rightarrow$ **8** $\rightarrow$ **5** $\rightarrow$ **2**

$1 + \text{DP}(25)$

$1 + \text{DP}(17)$

```
                                          8      5       2
                                         / \     |     / | \
                                        1  -7    2   -5 -13 -1
                                       /
                                     -2
```

Mohid : Why are we adding 1?
Prof : We are adding one because we are saying we used one coin.

Example

If we remove 2 from 17 (i.e., 17 - 2), we are left with 15. Then, by using the coin of 3 cents five times (since $3 \times 5 = 15$), we can fully cover the remaining 15 cents, solving the problem with a total of 6 coins: one 2-cent coin and five 3-cent coins.

-DP is only helpful in the case when there is something overlapping.

Recurrence in DP form

memo[i] represents the minimum number of coins required to make the value i.

Recurrence Relation in Dynamic Programming:
Base Case:
$memo[0] = 0$ and $memo[-k] = \infty$

Recurrence Relation:
We need to fill this till memo[-Cn-1]

$$memo[1] = \min \begin{cases} 1 + 1 - C_1 \\ 1 + 1 - C_2 \\ \vdots \\ 1 + 1 - C_n \end{cases}$$

Replace 1 with i

$$memo[i] = \min \begin{cases} 1 + i - C_1 \\ 1 + i - C_2 \\ \vdots \\ 1 + i - C_n \end{cases}$$

Topic 2:

Longest Increasing Subsequence Problem

Longest Increasing Subsequence

- **Subsequence:** It can be ordered but maybe not contiguous.
- Something super nice about increasing and decreasing subsequence.
- **Digression Theorem by Erdős–Szekeres:**
  - *"Any sequence of length n contains an increasing or a decreasing subsequence of size $\sqrt{n}$."*
- For 100 numbers, put in whatever sequence or order, there would be an increasing subsequence of length 10. If not increasing, then there would be a decreasing order — this is a consequence of something called Ramsey Theory.
- One of our seniors is doing PhD in Ramsey Theory.
- **Goal:** To find the longest increasing subsequence in a given sequence A.

Example:

$$[5, -10, -7, 3, 15, 12, 0, 9]$$

Longest increasing subsequence is: $-10, -7, 3, 15$; size: 4.

Initial Discussion

- **Prof:** Anyone wants to help me for solution to this?
- **Raveed:** Every number is the increasing subsequence of length 1. Start at a number and calculate the size of the increasing subsequence (max size) at that number by looking at previous numbers.
- For $A_i \leq n$:

$$\text{memo}[i] = \text{Longest increasing subsequence ending at index } i$$

- **Mohid:** "Can we use max?"
- **Prof:** Now we gonna define it recursively that Mohid was saying.

Dynamic Programming Approach

- How to compute memo$[i]$ through dynamic programming?
- Assuming we have all the memo up till i.
- If we have memo$[1]$, memo$[2]$, ..., memo$[i-1]$, then how to find memo$[i]$?
- **Example:**

$$[5, -10, -7, 3, 15, 12, 0, 9]$$

- Memo values so far: $1, 1, 2, 3,\ ?$
- What would be the memo$[i]$ at 15?

Finding memo[i]

- **Hamza:** If there is any smaller number before this number, then increase the size; otherwise, it remains the same.
- **Prof:** True. It's a max of things.
- We need to include memo[j]; memo of [j] would be $1, 1, 2, 3$.
- Need to find out where we add one and where we don't.
- **Khadijah:** If *i*th element is greater than previous, then we add one.
- **Prof:** Yes, for each of them, we check if 15 is greater than them.

$$\text{memo}[i] = \begin{cases} \text{memo}[j] + 1 & \text{if } A[j] \leq A[i] \\ \text{memo}[j] + 0 & \text{if } A[j] > A[i] \end{cases}$$

Example Continued

- So for memo[i] at 15:
$$\max\{2, 2, 3, 4\} = 4$$
- **Khadijah:** Please solve it for 12.
- **Professor:** For sure.
- For 12:
$$\max(2, 2, 3, 4, 4) = 4$$
- For 0:
$$\max(1, 2, 3, 3, 4) = 4$$
- For 9:
$$\max(1, 2, 3, 4, 4, 5) = 5$$

Ops! Something wrong has happened

Issue Identified

- **Professor:** A[i] should be the part of it.
- We modify it:

$$\text{memo}[i] = \text{LIS ending at } i \text{ and including } A[i]$$

- **Wasif:** Calculation for 0 is wrong; it should not be 4.
- **Prof:** We implemented according to algorithm, we are going to correct it.
- **Hamza:** If we change condition with 1, it's gonna work fine.
- **Prof:** Let's see.

New Approach for Longest Increasing Subsequence

New Approach:
memo[i] = LIS ending at i and including A[i]
Consider the sequence:
$[5, -10, -7, 3, 15, 12, 0, 9]$
We will now calculate the LIS for each element, including the element itself:

Index	0	1	2	3	4	5	6	7
Value	5	-10	-7	3	15	12	0	9
memo	1	1	2	?	?	?	?	?

Step-by-step calculation:
For all i, compute $\max\{1, \text{if } A[i] \geq A[j] \text{ then } \text{memo}[j] + 1 \text{ else } 0\}$
Final Algorithm:

- For all i:
$$\text{memo}[i] = \max\left(\{\text{memo}[j] + 1 \text{ if } A[i] \geq A[j] \text{ else } 0\}\right)$$
- Return the maximum value of $\text{memo}[i]$

Applying New Approach

- For 3:

$$\max(0+1, 1+1, 2+1, 1) = 3$$

- For 15:

$$\max(1+1, 1+1, 2+1, 3+1, 1) = 4$$

- For 12:

$$\max(1+1, 1+1, 2+1, 3+1, 0, 1) = 4$$

- For 0:

$$\max(0+1, 1+1, 2+1, 0, 0, 0, 1) = 3$$

- For 9:

$$\max(1+1, 1+1, 2+1, 3+1, 0, 0, 3+1, 1) = 4$$

Final Result

$$[5, -10, -7, 3, 15, 12, 0, 9]$$

Memo array:

$$1,\ 1,\ 2,\ 3,\ 4,\ 4,\ 3,\ 4$$

Hence, **LIS is 4**.

Clarifications

- **Khadijah:** Why are we doing 4 on 12 instead of 1?
- **Prof:** Because there is a subsequence which ends at 12, includes 12, and has length 4, i.e., $-10, -7, 3, 12$.
- **Prof:** Alright.

Professor: Ask questions or we move on to a new topic that is NP Hardness — no one asked though!

Topic 3:

NP Hardness

NP-Hardness: Part 1

Story Behind NP-Hardness:
Most of the algorithms we have studied in class are such that, if given a reasonably sized input, classical computers can solve them in seconds or at most a few hours. Around the 1950s and 60s, researchers began to notice certain problems for which no known algorithm could solve even moderately large instances within a practical timeframe.

For example, consider the **Longest Path Problem**. Given a graph $G(V, E)$ with 1000 vertices, there was no algorithm that could provide an answer within a reasonable amount of time.

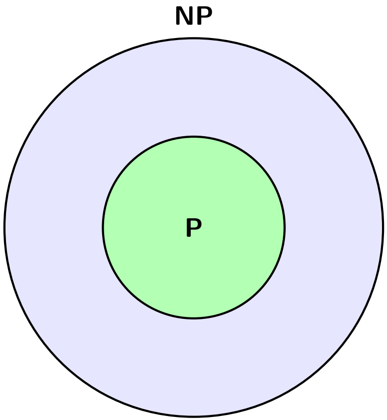
NP-Hardness: Part 2

This led to a dichotomy in algorithmic problems: some could be solved in **polynomial time**, i.e., in $O(n^k)$ time where k is a constant independent of the input size.
However, for problems like the Longest Path, the best known algorithms had exponential time complexity, like $O(2^n)$.

Yet, researchers couldn't prove that no polynomial-time algorithm exists for these problems. If they could, the story would have ended there.
But instead, more such difficult problems emerged — including the **Subset Sum Problem**, the **3-SAT Problem**, and even the Longest Path Problem with weight 1 on each vertex.

This gave rise to the classification of problems. Problems that can be solved in polynomial time were grouped into a class called **P**. Another broader class, called **NP** (Non-deterministic Polynomial time), was formed, which includes P as a subset.

Visualizing P and NP



- **P:** Problems that can be solved in polynomial time.
- **NP:** Problems whose solutions can be *verified* in polynomial time.
- All problems in P are also in NP, hence $\mathbf{P} \subseteq \mathbf{NP}$.

NP: Non-Deterministic Polynomial

Class of problems for which you can verify a "Yes" solution in polynomial time!

"At this point, the professor knew we weren't understanding, so he further elaborated:"

- Transform all problems into **decision problems**.
- That means: the algorithmic problem must only answer in binary – **Yes or No**.
- Can we do this for all problems we know?
- **Example (Shortest Path Problem):**
  - Original: "Give me the shortest path length from start to end."
  - Transformed: "Does a path from start to end exist with length less than 15?"
- Same idea works for the Longest Path Problem.

Understanding Verification in NP

- **NP class** contains problems for which a "Yes" answer can be **verified in polynomial time**.
- If the answer is "No", we don't care — but for "Yes", we must be able to verify it quickly.

- **Shortest Path (Decision Problem):**
  - If the answer is "Yes", the professor can ask: "Show me the path."
  - Then verify its length in polynomial time.

- **Distinction:**
  - **Polynomial:** Find and verify solution in polynomial time.
  - **Non-deterministic Polynomial:** Can verify "Yes" solution, not necessarily find it.

Certificate and Two-Party Verification

- **“Sometimes it’s easier to verify the solution than to find it.”**
- **Example (Longest Path Problem):**
  - We don’t know how to find the longest path efficiently.
  - But if someone gives one, we can check it in polynomial time.
- For verification, we need a **certificate**.
  - Any memory or solution of size $O(n^k)$.
- Think of it as a two-party game:
  - You = Challenger, Other = Prover
  - You give a decision problem.
  - If prover says “Yes”, you ask for a polynomial-sized certificate.
  - You can demand anything — but it must be verifiable within polynomial time.