# Lesson 6: Greedy Algorithms and Design Principles

A greedy algorithm is an approach used in computer science to solve optimization problems. It involves making choices that seem best at each step of the problem-solving process, without worrying about the potential consequences down the line. This method aims to find a solution that is optimal in a local sense, hoping that these local choices will ultimately lead to a global optimum. Greedy algorithms are particularly useful when dealing with problems where finding the best possible solution from a set of options is essential.

These algorithms are characterized by a few key features. Firstly, they make what's called a "greedy choice" at each step. This means that they select the most favorable option available at that specific moment. This choice should be feasible and help to break down the original problem into smaller, more manageable subproblems. Importantly, once a choice is made, it's considered final and there's no going back to revise it, which is known as the "greedy property."

Another crucial aspect is the concept of "optimal substructure." This means that the larger problem can be broken down into smaller subproblems, and solving these subproblems optimally can lead to an optimal solution for the overall problem. However, it's important to note that although greedy algorithms are often efficient, they don't always guarantee finding the best possible solution for every problem. Instead, they frequently offer good approximations for specific types of problems.

Several real-world examples illustrate the use of greedy algorithms. One instance is the "coin change problem," where the goal is to make change for a given amount using the fewest coins possible. Another is the "fractional knapsack problem," in which a set of items with varying weights and values need to be selected for a knapsack with a maximum weight capacity, aiming to maximize total value.

Despite their advantages, greedy algorithms also come with limitations. They don't guarantee the best result in all cases, which is a significant drawback. Careful consideration is needed to ensure that the locally optimal choices made at each step align with the desired global outcome. Analyzing the correctness and optimality of greedy algorithms can be complex and requires a good understanding of the problem at hand.

In summary, greedy algorithms are a practical approach in computer science to tackle optimization problems. While they might not always deliver the globally optimal solution,

they are valuable for their simplicity and efficiency in solving specific types of problems that adhere to the greedy choice and optimal substructure properties.

## Greedy choice property and optimal substructure

The "greedy choice property" and "optimal substructure" are fundamental concepts that underlie the effectiveness of greedy algorithms in solving optimization problems.

The **greedy choice property** refers to the principle that a greedy algorithm makes decisions based on what seems best at the current step, without considering the future consequences of those choices. In other words, it chooses the most favorable option available at the moment, hoping that this local choice will lead to a globally optimal solution. This property simplifies the decision-making process and is a hallmark of greedy algorithms.

On the other hand, **optimal substructure** is a property that allows a problem to be broken down into smaller, related subproblems. Solving these subproblems optimally contributes to finding an optimal solution for the original problem. This property is crucial for building a solution incrementally by making locally optimal choices. The optimal substructure property enables the recursive nature of many greedy algorithms, where solving smaller subproblems leads to solving the larger problem.

The interaction between these two properties is what empowers greedy algorithms to find approximate solutions efficiently. The greedy choice property guides the algorithm to make immediate choices that improve the solution step by step. Meanwhile, the optimal substructure property ensures that these local choices collectively contribute to an overall optimal solution.

However, it's important to note that not all problems possess these properties, making them unsuitable for greedy algorithms. In some cases, making decisions solely based on immediate benefits may lead to incorrect or suboptimal solutions. Therefore, careful consideration is necessary when applying greedy algorithms, especially in scenarios where the problem lacks the necessary properties for this approach to be effective.

The greedy choice property and optimal substructure are two core concepts that define the behavior of greedy algorithms. The former guides the algorithm to make locally optimal choices at each step, while the latter allows the problem to be divided into smaller subproblems whose solutions contribute to an overall optimal solution. These

properties, when present, make greedy algorithms a powerful tool for solving optimization problems efficiently.

# Examples of greedy algorithms

Certainly, here are explanations of the examples you mentioned: the Coin Change Problem, Huffman Coding, and the Activity Selection Problem, all of which can be solved using greedy algorithms.

## Coin Change Problem:

The Coin Change Problem is a renowned optimization challenge with practical implications in various financial scenarios, such as currency conversion and transactions. It involves working with a set of distinct coin denominations and a predefined target amount. The objective here is to develop a strategy that efficiently minimizes the number of coins needed to accurately represent the specified amount.

This problem holds substantial relevance in everyday financial operations where efficient algorithms for making change are crucial. By delving into the intricacies of the Coin Change Problem, we gain insights into the significance of algorithmic optimization in addressing real-world currency-related tasks.

**Problem Setting and Objective**
At its core, the problem entails an array of coin denominations, each signifying a distinct value (e.g., 1 cent, 5 cents, 25 cents), alongside a given target amount. The primary goal is to determine the smallest possible number of coins required to represent the target amount accurately using the provided denominations.

**Greedy Algorithm Approach**
A pragmatic approach to solving the Coin Change Problem is by employing a greedy algorithm. These algorithms make decisions at each step that seem optimal locally, with the anticipation that these choices will collectively yield a globally optimal outcome. In the context of this problem, the "greedy choice" entails consistently selecting the highest coin denomination that doesn't exceed the remaining target amount. This strategy prioritizes coins with the highest values.

*The procedural breakdown is as follows:*

1. Initiate the process by arranging the coin denominations in descending order, streamlining the selection of the highest-value coin.
2. Establish a variable, often termed "count," to monitor the count of coins used.
3. Begin with the highest coin denomination. Subtract its value from the target amount.
4. Repeat this sequence iteratively: at each step, opt for the largest coin denomination that fits within the remaining target amount, deduct its value, and increment the coin count.
5. Persist in this sequence until the target amount reaches zero.

The inherent advantage of this approach lies in its natural inclination to exhaust higher-value coins before resorting to smaller denominations. This "largest first" strategy significantly reduces the total number of coins needed to reach the target amount.

Nonetheless, it's essential to acknowledge that while the greedy approach works well for conventional coin systems, it might not universally apply. Instances involving unique coin sets or specific denominations might necessitate more intricate algorithms like dynamic programming to ensure an optimal solution. Therefore, while the greedy approach offers an efficient solution to the Coin Change Problem, its suitability should be assessed based on the specifics of the coin denominations and the problem context in question.

## Huffman Coding:

Huffman Coding stands as a cornerstone compression technique, playing a pivotal role in data encoding and transmission efficiency. This method undertakes the task of encoding characters from a data source in a manner that curtails the total bit count required. The outcome is a compact representation that greatly diminishes data size, making storage and transmission more resource-efficient.

**Compression and Efficient Data Handling**
In various practical scenarios, data files can be voluminous, making efficient storage and transmission paramount. Huffman Coding tackles this challenge by assigning variable-length binary codes to characters based on their occurrence frequencies in the source data. Characters that appear frequently receive shorter codes, optimizing the overall encoding process. This approach curtails the need for uniform-length codes and tailors the encoding scheme to the inherent distribution of characters within the data.

**Greedy Approach and Optimal Compression**

The central tenet of Huffman Coding is its utilization of a greedy algorithmic strategy. This strategy pursues immediate optimal choices in the hope of attaining a globally efficient outcome. The process initiates by assembling a priority queue containing individual characters and their respective frequencies. The algorithm then proceeds to iteratively combine the two least frequent characters into a single node of a binary tree, with the combined frequency serving as the frequency of the new node. This approach is repeated until only one node remains, resulting in a binary tree known as the Huffman Tree.

**Optimal Coding Scheme**
The brilliance of the Huffman Tree lies in its construction that ensures an optimal coding scheme. The path from the root of the tree to any character's leaf node corresponds to the binary code assigned to that character. Characters that occur more frequently are positioned higher in the tree, leading to shorter codes, while infrequent characters are located deeper in the tree, yielding longer codes. This design harmonizes with the goal of minimizing the total number of bits required for encoding the entire dataset.

Huffman Coding exemplifies the synergy of algorithmic ingenuity and data efficiency. By strategically tailoring codes to match the character distribution, this technique delivers substantial reductions in data size, proving indispensable in various applications ranging from data compression to network transmission. While its greedy approach might not always yield the globally optimal solution, in the case of Huffman Coding, it undeniably succeeds in producing a coding scheme that remarkably compresses data while maintaining the capacity for accurate decoding.

## Activity Selection Problem:

The Activity Selection Problem is a pivotal optimization puzzle that revolves around efficiently choosing a subset of activities from a provided collection, each with defined start and finish times. The primary objective is to construct the most extensive possible set of non-overlapping activities, thus optimizing the utilization of time slots.

**Optimizing Activity Scheduling:**
In various real-world scenarios, such as conference scheduling or project management, the selection of activities to maximize efficiency while preventing overlap is critical. The Activity Selection Problem serves as a formalized representation of this challenge. By solving it, we can effectively determine the most strategic way to schedule events, tasks, or assignments, optimizing resource allocation and productivity.

**Greedy Algorithm Approach:**
A common approach to addressing the Activity Selection Problem employs a greedy algorithm. Greedy algorithms capitalize on making locally optimal decisions at each step, with the overarching goal of achieving an optimal global outcome. In this context, the greedy choice involves arranging activities in order of their finish times. This arrangement facilitates the selection of activities with the earliest finishing times that do not clash with previously chosen activities.

*The procedure unfolds as follows:*

1. Sort the activities based on their finish times in ascending order.
2. Initialize a solution set to store the selected activities.
3. Begin with the activity that concludes earliest (lowest finish time) and add it to the solution set.
4. Iterate through the remaining activities: for each, select it if its start time is later than or equal to the finish time of the last selected activity.
5. Continue this process until all activities have been considered.

**Optimal Time Slot Utilization:**
The essence of the greedy approach here lies in its focus on immediate efficiency. By prioritizing activities that conclude earlier, the algorithm ensures that each chosen activity doesn't infringe upon the time slot of its predecessors. This leads to an assembly of non-overlapping activities, ultimately maximizing the utilization of available time intervals.

While the greedy approach is effective for this particular problem, its applicability depends on the problem's characteristics. In cases where the activities have complex dependencies or intricate constraints, alternative strategies like dynamic programming might be more suitable. Nonetheless, for scenarios where the objective is to make the most of available time slots through a straightforward approach, the Activity Selection Problem stands as a quintessential example of the power of greedy algorithms.

These examples illustrate how greedy algorithms work by making locally optimal choices at each step. However, it's important to note that while greedy algorithms work for these specific problems, they might not be suitable for all optimization problems. Careful consideration of the problem's characteristics and whether it exhibits the greedy choice property and optimal substructure is crucial when deciding whether to use a greedy algorithm.

# When to use greedy algorithms and their limitations

## When to Use Greedy Algorithms:

Greedy algorithms prove their worth in addressing optimization challenges that align with specific problem characteristics. Here are instances where the application of a greedy approach is advantageous:

- Greedy Choice Property: Greedy algorithms shine when the notion of making locally optimal choices at each step contributes to an optimal global solution. Problems that can be dissected into smaller subproblems, with each subproblem's solution playing a role in achieving the overall optimal solution, often align with the greedy approach.

- Optimal Substructure: A fertile ground for greedy algorithms is found in problems that can be partitioned into smaller subproblems, where the pursuit of an optimal solution for these subproblems harmonizes with the optimal solution for the larger problem.

- Immediate Selection Requirements: Situations that necessitate prompt decision-making, with no need to revisit past choices, are well-suited for greedy algorithms. These algorithms boast efficiency and are particularly handy when swift decisions are pivotal.

- Simplicity in Constraints: Greedy algorithms flourish when complexities, intricate constraints, or dependencies that might lead to intricate decision processes are absent. Problems governed by straightforward rules tend to harmonize with the straightforward nature of greedy solutions.

## Limitations of Greedy Algorithms:

Despite their prowess in specific contexts, greedy algorithms are not universally all-encompassing. Several limitations warrant careful consideration:

- Global Optimum Uncertainty: Greedy algorithms lean towards local optimization, which does not invariably guarantee the discovery of the globally optimal solution. Certain cases could see an initial locally optimal choice culminate in an overall suboptimal outcome.

- Complex Dependencies: Problems entailing intricate interdependencies among decisions can stymie the effectiveness of a greedy approach. Such scenarios can elude the broader context, potentially prompting selections that appear ideal in isolation but falter in a holistic context.

- Counterexamples: Some situations may initially appear amenable to a greedy solution, yet deeper scrutiny could unveil counterexamples where this approach yields incorrect results.

- Proof of Correctness: Demonstrating the correctness of a greedy algorithm poses a challenge. Establishing that the greedy choice property and optimal substructure indeed hold demands meticulous analysis.

- Problem Variability: Variants of a problem can render a once-viable greedy approach obsolete. Modifying problem constraints or introducing additional criteria may disrupt the feasibility of an initially fitting greedy solution.

In summation, the prowess of greedy algorithms lies in their adeptness at solving optimization problems tailored to the greedy choice property and optimal substructure. The swiftness and efficiency they offer are undeniable assets. Nevertheless, the limitations they carry—such as the absence of a universal global optimality assurance and their inadequacy for intricate dependency-riddled scenarios—emphasize the need for prudent consideration of problem attributes prior to embracing a greedy methodology.

## Comparing dynamic programming and greedy approaches

In the realm of optimization problem-solving, two prominent strategies, dynamic programming and greedy algorithms, stand out with their distinct attributes and applications. A comparative analysis between these two methods sheds light on their strengths and limitations, revealing the contexts in which each approach shines.

### 1. Nature of Solutions:

**Dynamic Programming:** A hallmark of dynamic programming is its systematic quest for the optimal solution by deconstructing the problem into smaller, manageable subproblems. This approach operates on the principle of reusing previously computed solutions, building a cumulative solution step by step. The method ensures

comprehensive exploration of subproblems, culminating in a final solution that is inherently optimal through its consideration of all potential avenues.

**Greedy Algorithms:** In contrast, greedy algorithms operate on the philosophy of immediate gratification. At each juncture, they make choices that seem locally optimal, banking on the belief that these choices will collectively lead to a global optimum. This strategy prioritizes the most favorable choice at each step without concern for potential downstream implications. Although this approach doesn't guarantee a globally optimal solution, it excels in its efficiency and efficacy under specific circumstances.

## 2. Optimal Substructure:

**Dynamic Programming:** Dynamic programming finds its stride when dealing with problems that boast optimal substructure. These are problems where breaking them down into subproblems leads to a coherent path to the overall solution. The method employs a bottom-up construction, synthesizing solutions to smaller subproblems to craft a final solution. The interconnection between subproblems ensures that the whole is greater than the sum of its parts.

**Greedy Algorithms:** Greedy algorithms are a natural fit for problems characterized by the greedy choice property. This property dictates that making immediate locally optimal selections translates to achieving global optimality. Greedy algorithms operate top-down, focusing their attention on current optimization without dwelling on the broader implications, making them suitable for select contexts.

## 3. Overlapping Subproblems:

**Dynamic Programming:** An area where dynamic programming thrives is in problems replete with overlapping subproblems. The method capitalizes on this repetition by storing solutions to these subproblems, reducing redundant computations and enhancing efficiency. By leveraging previously computed solutions, dynamic programming gains momentum in its quest for the optimal solution.

**Greedy Algorithms:** Unlike dynamic programming, the effectiveness of greedy algorithms is not intrinsically linked to overlapping subproblems. These algorithms are guided by the immediate rewards of their choices rather than the potential for reusing solutions.

## 4. Time Complexity:

**Dynamic Programming:** The robust nature of dynamic programming often results in higher time complexity, necessitating the solution of all conceivable subproblems. Strategies like memoization and tabulation are employed to mitigate this complexity and streamline the process.

**Greedy Algorithms:** Greedy algorithms showcase efficiency, thanks to their instantaneous decision-making. By bypassing the need to revisit prior choices, they often lead to expedited approximations of solutions.

## 5. Guarantee of Optimality:

**Dynamic Programming:** Dynamic programming exudes confidence in its guarantee of locating the globally optimal solution. Through its systematic exploration of all possibilities, it leaves no stone unturned, providing assurance that the best outcome is attained.

**Greedy Algorithms:** Greedy algorithms, while rapid and effective in their approach, do not carry the same guarantee. Their focus on local optimization may occasionally disregard superior choices, making them more prone to approximations rather than certainties.

In summation, the selection between dynamic programming and greedy algorithms hinges on the intricacies of the problem at hand. Dynamic programming assures optimality while handling optimal substructure and overlapping subproblems. On the other hand, greedy algorithms are renowned for their efficiency, finding their forte in problems characterized by the greedy choice property. The art of selecting the right approach is rooted in comprehending the specific requirements and constraints of the problem, ultimately determining whether the path to a solution aligns more harmoniously with the systematic thoroughness of dynamic programming or the swift, immediate gratification of greedy algorithms.