Research for Greedy Algorithm

What is greedy algorithm?

A greedy algorithm is an algorithm that finds a solution to problems in the shortest time possible. It picks the path that seems optimal at the moment without regard for the overall optimization of the solution that would be formed.

A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result.

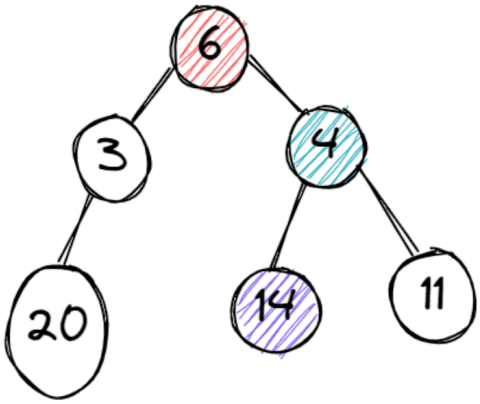
The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top-down approach.

How do you know that is a greedy algorithm?

An algorithm is greedy when the path picked is regarded as the best option based on a specific criterion without considering future consequences. But it typically evaluates feasibility before making a final decision. The correctness of the solution depends on the problem and criteria used.

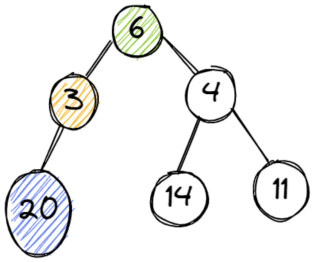
Example: A graph has various weights and you are to determine the maximum value in the tree. You'd start by searching each node and checking its weight to see if it is the largest value.

Greedy vs Not Greedy Algorithms



A greedy algorithm will solve this problem in the following way:
Starting from vertex 6, then it's faced with two decisions – which is bigger, 3 or 4? The algorithm picks 4, and then is faced with another decision – which is bigger, 14 or 11. It selects 14, and the algorithm ends.

On the other hand there is a vertex labeled 20 but it is attached to vertex 3 which greedy does not consider as the best choice. It is important to select appropriate criteria for making each immediate decision.



A non-greedy algorithm would solve this problem in the following manner:

Starting from vertex 6, then it's faced with two decisions – which is bigger, 3 or 4? The algorithm picks 4, and then is faced with another decision – which is bigger, 14 or 11. It selects 14 and keeps it aside.

Then it runs the process again, starting from vertex 6. It selects the vertex with 3 and checks it. 20 is attached to the vertex 3 and the process stops. Now it compares the two results – 20 and 14. 20 is bigger, so it selects the vertex (3) that carries the largest number and the process ends.

Algorithm design techniques are strategies or approaches used to develop efficient and effective algorithms for solving computational problems. Here are some fundamental algorithm design techniques:

1. Greedy Algorithms:

Greedy algorithms make locally optimal choices at each step with the hope of finding a global optimum. They are often intuitive and simple to implement. Examples include Dijkstra's algorithm for shortest paths and Huffman coding for data compression.

2. Divide and Conquer:

Divide and conquer involves breaking down a problem into smaller sub-problems, solving them recursively, and combining their solutions to solve the original problem. Classic examples include merge sort and quicksort for sorting, and the fast Fourier transform (FFT) for signal processing.

3. Dynamic Programming:

Dynamic programming is an optimization technique where a problem is broken down into overlapping subproblems, and the solutions to these subproblems are stored to avoid redundant computations. Common examples include the knapsack problem and the Floyd-Warshall algorithm for all-pairs shortest paths.

4. Backtracking:

Backtracking is a systematic way of trying out different possibilities and undoing those choices that do not lead to a solution. It is often used for problems where solutions are built incrementally, such as the N-Queens problem or the traveling salesman problem.

5. Brute Force:

Brute force is a straightforward approach that systematically enumerates all possible solutions and checks each one to determine if it satisfies the problem's constraints. While not always efficient, it provides a simple baseline. Examples include the brute force approach to the traveling salesman problem.

However, we can determine if the algorithm can be used with any problem if the problem has the following properties:

1. Greedy Choice Property

If an optimal solution to the problem can be found by choosing the best choice at each step without reconsidering the previous steps once chosen, the problem can be solved using a greedy approach. This property is called greedy choice property.

2. Optimal Substructure

If the optimal overall solution to the problem corresponds to the optimal solution to its subproblems, then the problem can be solved using a greedy approach. This property is called optimal substructure.

The greedy approach has several advantages in algorithm design, making it a popular choice for solving certain types of problems. Here are some of the key advantages:

1. Simplicity:
- Greedy algorithms are often simple and easy to understand. The straightforward nature of the greedy approach makes it accessible for quick implementation and comprehension.

2. Efficiency:
- Greedy algorithms are typically efficient in terms of time complexity. They often have linear or close-to-linear time complexities, making them suitable for large datasets.

3. Local Optima:
- The greedy approach makes decisions based on local optimality at each step. While this doesn't guarantee a globally optimal solution, it often leads to solutions that are close to optimal. In many cases, a locally optimal choice at each step results in a globally optimal solution.

Implementing a greedy algorithm for job scheduling in a web app

1. Define the Problem:

Understand the job scheduling problem you're dealing with. Define constraints, goals, and parameters.

2. Design Algorithm Logic:

For a greedy approach:
- Identify Criteria: Determine the criteria to prioritize jobs (e.g., shortest duration, earliest deadline).
- Sort Jobs: Sort the list of jobs based on the chosen criteria.
- Iterate & Select: Iterate through the sorted list, selecting jobs that meet criteria until resources are exhausted or scheduling is complete.

3. Implement the Algorithm:

Use a programming language (JavaScript, Python, etc.) for the web app. Incorporate the algorithm logic into the app's backend or frontend.

5. Testing and Optimization:

Test the algorithm with various scenarios to ensure it performs as expected. Consider optimizations based on specific use cases or edge cases.

6. User Interface:

Design the user interface to input job details and display the scheduled jobs. Provide a clear and intuitive way for users to interact with the scheduling system.

7. Error Handling & Edge Cases:

Consider scenarios where the algorithm might fail or produce suboptimal results and implement error handling or adjustments to address these cases.

8. Deployment and Maintenance:

Deploy the web app and periodically maintain it to ensure smooth functioning and consider user feedback for improvements.

Remember, job scheduling problems can vary significantly, and the specific details of your scenario might require adjustments or enhancements to this basic greedy algorithm implementation.