# Greedy Algorithm

## Algorithm

1. Determine ratio/unit value, e.g.
2. Sort by **[increasing]** / **[decreasing]** order
3. Select the first item and repeat until constraint is met

## Proof via Exchange Argument

Suppose:

* is the greedy solution.
* is some optimal (possibly different) solution.

**Step 1:** Show that and may differ at some step (say, the first step they disagree).

**Step 2:** Exchange the element in with the one in at that step.

**Step 3:** Show that this swap does not reduce the quality of and repeat the process until .

⇒ Therefore, must be optimal too.

---

**DIVIDE AND CONQUER**

## Algorithm

1. Divide the input into smaller subproblems.
2. Conquer them recursively.
3. Combine/Compare the results to determine a final answer.

## Proof via Induction

**Base Case:** Show that the algorithm works correctly on the smallest input (e.g. single element array).

**Inductive Hypothesis:** Assume the algorithm works correctly for inputs of size .

**Inductive Step:** Show that for an input of size :

* The algorithm divides the input into one or more strictly smaller subproblems of size
* By the inductive hypothesis, these smaller subproblems are solved correctly.
* The algorithm then combines the solutions of the subproblems into a correct overall solution.

⇒ By mathematical induction, the algorithm works for all input sizes.

---

**Master Theorem**

Where:

# DYNAMIC PROGRAMMING

## Algorithm

1. Define the subproblem that needs to be solved repeatedly (e.g. let optimal solution with coins).
2. Determine the main cases in terms of smaller sub problems (e.g. ).
3. Tabulate the results of subproblems (e.g. ).
4. Compute solution.

## Proof via Induction op

**Base Case:** Show that the value of the base subproblem (e.g. is computed correctly).

**Inductive Hypothesis:** Assume that all subproblem values are correctly computed by the recurrence.

**Inductive Step:** Show that for :

* The recurrence uses only previously computed values (≤ n), which are correct by the hypothesis.
* The maximum or minimum rule, or other operation, is applied correctly.
* Therefore, is computed correctly.

⇒ By mathematical induction, all sub problems, (e.g. ) are correct, and the final algorithm is correct.

---

**nondeterministic Polynomial**

A decision problem is in NP if you can verify it in polynomial time.

1. Describe the format of a proposed solution.
2. Show how to verify the certificate in polynomial time.

---

**NP-Hard**

A problem is NP-Hard if every problem in NP can be reduced to it in polynomial time.

To prove a problem is NP-Hard:

1. Choose a known NP-Complete problem (like SAT, 3-SAT).
2. Reduce it to your problem in polynomial time.
3. You do not need to show your problem is in NP.

---

**NP-complete**

To prove a problem is NP-Complete:

1. Show it’s in NP
2. Reduce a known NP-Complete problem to it in polynomial time: (e.g., reduce 3-SAT or Vertex Cover to your problem).

---

**Disprove Correctness**

To disprove the correctness of **ANY** algorithm, you **MUST** provide a counterexample

**Network flow**

## Algorithm (Ford–Fulkerson Method)

Given a valid graph (source + sink nodes + edge weights):

1. Create a residual graph and
2. While there exists an augmenting path from source to sink
   1. Find a path using DFS, BFS, or heuristics.
   2. Determine the minimum residual capacity along this path.
   3. Augment flow along the path by , updating residual capacities and add to .
3. Repeat until no augmenting path exists and .

## Proof of Correctness

At each iteration:

* No edge exceeds its capacity.
* The flow respects flow conservation.

**Termination (Integer Capacities)**:

* If all capacities are integers, each augmentation increases flow by **at least 1**.
* Since total flow is bounded, the algorithm terminates in **finite steps**.

**Correctness Guarantee**:

* When no augmenting path remains, the current flow is:
  + **Feasible** (obeys all constraints), and
  + **Maximum**, by the **Max-Flow Min-Cut Theorem**:

Maximum flow value = Minimum capacity of any s-t cut.

⇒ Therefore, **SUM = max flow**, and the algorithm is correct.

---

**Reduction**

## Edmond Karp

You convert Problem A into a single instance of Problem B (in polynomial time) and then solve B once.

## Cook Levin

You solve Problem A by asking for help multiple times from a known Problem B (in NP), treating B like a black box.

---

**Known NP-Complete Problems**

**SAT (Boolean Satisfiability)**

Given a Boolean formula, is it satisfiable? (does there exist an assignment such that the output returns true)

**3-SAT**

Same as SAT, but each clause has exactly 3 literals.

**CLIQUE**

Given a graph and integer , does it have a clique of size ? A clique is a complete subgraph — all the nodes in the group are directly connected to each other.

**VERTEX COVER**

Given a graph and an integer , can you choose vertices that touch all edges?

**INDEPENDENT SET**

Does the graph contain vertices with no edges between them?

**HAMILTONIAN CYCLE**

Is there a cycle that visits every vertex exactly once?

**GRAPH COLORING (k-colouring)**

Can you colour the graph using colours such that no adjacent nodes share the same color?

**FEEDBACK VERTEX SET**

Remove the fewest vertices to make the graph acyclic.

**KNAPSACK (0/1 version)**

Choose items with maximum value without exceeding weight limit.

**SUBSET SUM**

Is there a subset of numbers that adds up to a given target?

**PARTITION**

Can a set of integers be split into two subsets with equal sum?

**JOB SCHEDULING (with deadlines or resource constraints)**

Can tasks be scheduled with limited resources or time?

**TRAVELING SALESMAN PROBLEM (TSP) (Decision version)**

Is there a route of cost that visits all cities exactly once?

**EXACT COVER / SET COVER**

Can a collection of sets cover all elements using exactly one element from each set?

**LONGEST PATH**

Given two vertices, is there a path of at least length between them?

---

**Sample Karp Reduction**

Given a 3-SAT formula

We create 1 node for each literal in each clause and connect nodes from different clauses if their literals are non-contradictory (e.g. and are contradictory → don’t connect them) and set .

Does this graph contain a 2-clique?

If YES → the original formula is satisfiable

If NO → the formula is unsatisfiable