Recipe

1 Greedy

1.1 Greedy Stays Ahead

1.1.1 Define Your Solution

Your algorithm will produce some object G and you will probably compare it against optimal solution O. Introduce some variables denoting your algorithm's solution and the optimal solution.

1.1.2 Define Your Measure

Your goal is to find a series of measurements you can make of your solution and the optimal solution. Define some series of measures $m_1(G), m_2(G), ..., m_k(G)$ such that $m_1(O), m_2(O), ..., m_n(O)$ is also defined for some choices of m and n.

1.1.3 Prove Greedy Stays Ahead

Prove that $m_i(G) \geq m_i(O)$ or that $m_i(G) \leq m_i(O)$, whichever is appropriate, for all reasonable values of i. This argument is usually done inductively.

1.1.4 Prove Optimality

Using the fact that greedy stays ahead, prove that the greedy algorithm must produce an optimal solution. This argument is often done by contradiction by assuming the greedy solution is not optimal (the relationship between n and k) and using the fact that greedy stays ahead to derive a contradiction.

1.2 Exchange Arguments

Show that you can iteratively transform any optimal solution into the solution produced by the greedy algorithm without worsening the cost of the optimal solution, thereby proving that the greedy solution is optimal.

1.2.1 Define Your Solutions

You will be comparing your greedy solution G to an optimal solution O.

1.2.2 Compare Solutions

Show that if $G \neq O$, then they must differ in some way. This could mean that there is a piece of G that is not in O, or that two elements of G are in a different order in O.

1.2.3 Exchange Pieces

Show how to transform O by exchanging some piece of O for some piece of G. Then prove that by doing so, you did not worsen the quality of O and therefore have a different optimal solution.

1.2.4 Iterate

Argue that you have decreased the number of differences between G and O by performing the exchange, and that by iterating this process for a finite number of times you can turn O into G without impacting the quality of the solution. Therefore, G must be optimal.

2 Dynamic Programming

The general approach of the DP problem is:

1. Characterize structure of problem: identify **subproblems** whose optimal solutions can be used to build an optimal solution to original problem. Conversely, given an optimal solution to original problem, identify subparts of the solution that are optimal solutions for some subproblems.
2. Write the **recurrence** and **initial cases**, know where the solution of problem is.
3. Look at precedence constraints (draw a figure) and write the algorithm (iterative, or recursive with memos).
4. Study the problem complexity (straightforward with iterative algorithm; don't forget the time to compute one subproblem).
5. Construct optimal solution from computed information (back-tracing).

3 NP-Complete

Recipe to establish **NP-completeness** of problem Y:

1. **Step 1:** Show that Y is in NP:
   - Describe how a potential solution will be represented
   - Describe a procedure to check whether the potential solution is a correct solution to the problem instance, and argue that this procedure takes **polynomial time**.
2. **Step 2**: Choose an NP-complete problem X.

3. **Step 3**: Prove that $X \leq_P Y$ (X is poly-time reducible to Y):
    - Describe a procedure f that converts the inputs i of X to inputs of Y in polynomial time.
    - Show that the reduction is correct by showing that $X(i) = YES$ if and only if $Y(f(i)) = YES$
4. **Step 4**: Justification: If X is an NP-complete problem, and Y is a problem in NP with the property that $X \leq_P Y$, then Y is NP-complete.

The hardest part is step 3. The recipe for this step is shown below:

- Let I_1 be any instance of X.
- Transform I_1 into an instance I_2 of problem Y.
- Check whether this transformation takes a polynomial time.
- **Suppose** I_1 **has a solution**, then prove that I_2 also has a solution.
- **Suppose** I_2 **has a solution**, then show that it implies that I_1 has a solution.

4 Graph

4.1 Cut property

Assume all edge costs c_e are distinct. Then consider a cut in a graph that divides the vertices into two disjoint subsets. Among the edges that cross the cut (those that have one endpoint in each subset), if the weight of an edge e is the smallest, then **every MST** contains e.

4.2 Cycle property

Now, define C as any cycle, and let f be the max cost edge belonging to C. **Then every MST does not contain** f.

4.3 Prim's Algorithm (Cut)

The Prim's Algorithm includes the following steps:

1. Start with a set 'MST' that contains a chosen starting vertex (arbitrary)
2. While 'MST' does not yet include all vertices:
    (a) Select and remove the edge with the smallest weight that **connects a vertex in MST to a vertex outside MST**
    (b) Add the selected edge and vertex not in MST to MST
    (c) Update the priority queue by adding the new edges that connect the newly added vertex to any vertex not yet in MST

4.4 Kruskal's Algorithm (Cut, Cycle)

The Kruskal's Algorithm includes the following steps:

1. First sort all the edges of the graph in **non-decreasing** order of their weights.
2. Start with $T = \emptyset$. Insert edge e in ascending order in T unless doing so would create a cycle.

4.5 Reverse-Delete Algorithm (Cycle)

Start with $T = E$, where E is the edge array. Then consider edges in descending order of cost. Delete edge e from T unless doing so would disconnect T.