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.

5 Greedy Prove Graphs

5.1 Dijkstra (Greedy Stays Ahead)

To prove Dijkstra's algorithm is correct, we'll show that at each step of the algorithm, the distance values assigned to the vertices are optimal, meaning they represent the shortest possible distances from the source vertex s to the vertices processed so far.

Greedy Stays Ahead Principle

- **Invariant**: After each iteration of the main loop (i.e., after each extraction from the priority queue), the distances dist[u] for the extracted vertex u is the shortest path distance from the source s to u.
- This means that once a vertex u is removed from the priority queue, its shortest distance is finalized and will not change.

Base Case

- Initially, the source s is assigned dist[s] = 0, which is trivially correct since the distance from the source to itself is zero.
- All other vertices are initialized with an infinite distance, representing that they are initially unreachable.

Inductive Step

- Assume that after processing the first k vertices (i.e., after extracting k vertices from the priority queue), the shortest path distances for these vertices are correctly computed.
- Consider the next vertex u that is extracted from the priority queue. By the properties of the priority queue and the algorithm, u has the smallest dist[u] among all vertices still in the queue.

- Let $P(s, u)$ be the path from s to u with this computed shortest distance. Any other path P' from s to u would either:
  - Already have a larger computed distance for one of the vertices in P' before reaching u, or
  - Involve a vertex v that was not yet processed, meaning $\text{dist}[u] < \text{dist}[v]$ and hence u is reached earlier than v.
- Thus, the distance $\text{dist}[u]$ is indeed the shortest path from s to u, and once u is processed, this distance is finalized.

Conclusion

- The "greedy stays ahead" principle ensures that each time a vertex is selected and its shortest distance is finalized, the algorithm maintains the invariant that all finalized distances are correct.
- By the time all vertices have been processed, the shortest path distances for all vertices in the graph have been correctly computed.

5.2 Dijkstra (Exchange Argument)

To prove the correctness of Dijkstra's algorithm using the exchange argument, we'll assume that the algorithm has produced a solution (i.e., a set of shortest paths to each vertex) and then consider any alternative sequence of vertex selections to show that it cannot result in a better (i.e., shorter) set of paths.

Step 1: Assume an Alternative Optimal Sequence

- Let's assume there is an optimal solution that uses a different sequence of vertex selections compared to Dijkstra's algorithm.
- Let's denote this sequence by $O_1, O_2, \ldots, O_n$, where each O_i represents the vertex selected at step i in the alternative sequence.
- Let $G_1, G_2, \ldots, G_n$ be the sequence of vertex selections made by Dijkstra's algorithm.

We want to prove that the total distance of the paths found by Dijkstra's algorithm is no worse than the distance of the paths found by this alternative sequence.

Step 2: Consider the First Point of Difference

- Identify the first point in the sequence where the alternative optimal sequence $O_1, O_2, \ldots, O_n$ differs from Dijkstra's sequence $G_1, G_2, \ldots, G_n$.
- Let G_i be the vertex chosen by Dijkstra's algorithm at step i, and O_j be the vertex chosen by the alternative sequence at the same step i (where j is not necessarily equal to i).

Since this is the first difference in the sequences, we have $G_1 = O_1$, $G_2 = O_2$, ..., $G_{i-1} = O_{i-1}$.

Step 3: Exchange Argument

- Dijkstra's algorithm picks G_i because dist$[G_i]$ is the smallest distance among all vertices not yet processed. This means dist$[G_i] \leq$ dist$[O_j]$.
- If the alternative sequence picked O_j instead of G_i, then dist$[O_j]$ must be at least dist$[G_i]$. Thus, picking O_j cannot lead to a shorter path for any subsequent vertex.

Step 4: Iterative Argument

- Suppose we swap G_i and O_j in the alternative sequence. After this swap, we continue to compare the sequences at subsequent steps.
- If we continue this swapping process for every differing pair, each swap can only either:
  - Keep the total distance the same, or
  - Reduce the total distance in Dijkstra's favor, because Dijkstra's selection of the next vertex G_i is always based on the smallest distance.

As a result, by swapping all differences, the alternative sequence can be transformed into Dijkstra's sequence without increasing the total path length.

Step 5: Conclusion

- Since any alternative sequence can be transformed into Dijkstra's sequence through a series of swaps without increasing the total distance, this implies that Dijkstra's algorithm is at least as good as any other possible sequence in terms of producing the shortest paths.
- Therefore, Dijkstra's algorithm is optimal, and its greedy approach of selecting the vertex with the smallest tentative distance at each step ensures that the shortest paths are correctly calculated.

5.3 Prim's Algorithm (Greedy Stays Ahead)

To prove Prim's algorithm is correct, we'll show that at each step of the algorithm, the selection of edges and vertices ensures that the MST constructed so far is always optimal.

Greedy Stays Ahead Principle

- **Invariant**: After each iteration of the main loop (i.e., after each vertex is added to the MST set T), the tree T is a subtree of some MST for the graph.
- This means that the set T forms a connected subgraph that is part of an optimal MST.

Base Case

- Initially, the MST set T contains just the starting vertex r, which trivially forms a subtree of the MST.
- All other vertices are initialized with an infinite distance, representing that they are not yet connected to the tree.

Inductive Step

- Assume that after processing the first k vertices (i.e., after adding k vertices to T), the current tree T is a subtree of some MST.
- Consider the next vertex u that is extracted from the priority queue. By the properties of the priority queue and the algorithm, u is the closest vertex to the tree T, meaning the edge (u, v) connecting u to T has the smallest weight among all edges that could extend T.
- If there were a different MST that did not include the edge (u, v), we could replace an edge in that MST with (u, v) to get a new MST with a smaller or equal total weight, which contradicts the assumption that we already had the MST. Thus, u and the edge (u, v) must be part of the MST.

Conclusion

- The "greedy stays ahead" principle ensures that each time a vertex is added to the MST set T, the current tree T remains a subtree of the MST.
- By the time all vertices have been processed, the entire tree T is the MST for the graph.

5.4 Prim's Algorithm (Exchange Argument)

We will prove that the MST produced by Prim's algorithm is optimal by showing that any other MST can be transformed into the one produced by Prim's algorithm without increasing the total cost.

Step 1: Assume an Optimal MST

- Let T be the MST produced by Prim's algorithm.
- Let T^* be any other MST for the graph.

We aim to show that T and T^* have the same total weight, meaning Prim's algorithm produces an optimal MST.

Step 2: Identify the First Difference

- Suppose T and T^* differ in some edges. Let e be the first edge that is in T but not in T^*.
- Adding e to T^* would create a cycle because T^* is already a spanning tree.
- This cycle must contain some edge e' from T^* that is not in T.

Step 3: Edge Replacement

- Consider replacing e' in T^* with e. The result is a new tree $T' = T^* - e' + e$.
- Since e was chosen by Prim's algorithm, we know $w(e) \leq w(e')$. This means the total weight of T' is less than or equal to the total weight of T^*.

Step 4: Iterative Process

- We can repeat this edge replacement process, gradually transforming T^* into T without increasing the total weight.
- After enough replacements, T^* is transformed into T, showing that T is also an MST and has the same total weight as T^*.

Conclusion

- Since any MST T^* can be transformed into the MST T produced by Prim's algorithm without increasing the total weight, it follows that T is optimal.
- Therefore, Prim's algorithm produces the correct and optimal MST.

5.5 Kruskal's Algorithm (Greedy Stays Ahead)

To prove that Kruskal's algorithm correctly produces a Minimum Spanning Tree, we will use the **Greedy Stays Ahead** principle.

Define the Solutions

- Let T be the tree constructed by Kruskal's algorithm.
- Let T^* be any other MST.

We need to show that the total weight of T is at least as good as T^*, and hence T is an MST.

Greedy Stays Ahead Principle

- **Invariant**: At each step of Kruskal's algorithm, the set of edges E_T in the current tree T forms a subset of some MST. This means that each edge added to T is optimal given the choices made so far.

Base Case

- Initially, the MST T is empty, so the invariant holds trivially.

Inductive Step

- Suppose that after adding k edges, the set E_T is a subset of some MST. Now consider the $(k+1)$-th edge $e = (u, v)$ that Kruskal's algorithm adds to T. This edge is the smallest available edge that does not form a cycle in T.
- If e were not in T^*, adding it would create a cycle. This cycle must contain at least one other edge e' from T^* that is not in T.
- Since Kruskal's algorithm selects the smallest edge, $w(e) \leq w(e')$. We could replace e' with e in T^*, forming a new MST $T' = T^* - e' + e$ with a weight less than or equal to the original MST T^*.
- Therefore, after adding e, the new set E_T remains a subset of some MST.

Conclusion

- By the time all edges are processed and T is complete, the set of edges E_T forms an MST.
- Since Kruskal's algorithm maintains this invariant at each step, it follows that T is indeed a Minimum Spanning Tree.

5.6 Kruskal's Algorithm (Exchange Argument)

We will prove that the MST produced by Kruskal's algorithm is optimal by showing that any other MST can be transformed into the one produced by Kruskal's algorithm without increasing the total cost.

Step 1: Assume an Optimal MST

- Let T be the MST produced by Kruskal's algorithm.
- Let T^* be any other MST for the graph.

We aim to show that T and T^* have the same total weight, meaning Kruskal's algorithm produces an optimal MST.

Step 2: Identify the First Difference

- Suppose T and T^* differ in some edges. Let e be the first edge that is in T but not in T^*.
- Adding e to T^* would create a cycle because T^* is already a spanning tree.
- This cycle must contain some edge e' from T^* that is not in T.

Step 3: Edge Replacement

- Consider replacing e' in T^* with e. The result is a new tree $T' = T^* - e' + e$.
- Since e was chosen by Kruskal's algorithm, we know $w(e) \leq w(e')$. This means the total weight of T' is less than or equal to the total weight of T^*.

Step 4: Iterative Process

- We can repeat this edge replacement process, gradually transforming T^* into T without increasing the total weight.
- After enough replacements, T^* is transformed into T, showing that T is also an MST and has the same total weight as T^*.

Conclusion

- Since any MST T^* can be transformed into the MST T produced by Kruskal's algorithm without increasing the total weight, it follows that T is optimal.
- Therefore, Kruskal's algorithm produces the correct and optimal MST.