Divide and Conquer

Steps:

- divide: I into some number of smaller instances of the same problem p
- recurse: on each of the smaller instances to get the answer
- combine: the answers to produce an answer for the original instance I

Notes:

- Inductive proof
- Work and span by recurrences
- Naturally parallel

Contraction (Reduction, Transform and Conquer)

Steps:

- contract: "contract" (map) instance of problem p to smaller instance
- solve: solve smaller instance recursively
- expand: use the solution to solve original instance

Notes:

- Inductive proof
- Work by recursive (inductive) relation
- Efficient if reduce the problem size geometrically (constant factor < 1)

Dynamic Programming

Dynamic Programming:

Dynamic programming is a technique for solving problems recursively and is applicable when the computations of the subproblems overlap

DP Tools:

- Memoization (Top down)
  - an optimization technique where you cache previously computed results, and return the cached result when the same computation is needed again
  - storing the results of expensive function calls and returning the result when the same inputs occur again
- Tabulation (Bottom Up)
  - using iterative approach to solve the problem by solving the smaller sub- problems first and then using it during the execution of bigger problem
- Comparison
  - memoization usually requires more code and is less straightforward, but has computational advantages in some problems
    - mainly those which you do not need to compute all the values for the whole matrix to reach the answer
  - tabulation is more straightforward, but may compute unnecessary values
    - if you do need to compute all the values, this method is usually faster, though, because of the smaller overhead

Examples:

- Longest Common Subsequence problem
- Knapsack
- Travelling salesman problem

Sources:

- `http://stackoverflow.com/questions/12042356/memoization-or-tabulation-approach-for-dynami`

Greedy

Greedy:

- algorithm that makes the locally optimal choice at each stage
- a greedy algorithm never reconsiders its choices
  - choice made by a greedy algorithm may depend on choices made so far, but not on future choices or all the solutions to the subproblem

Greedy components:

- a candidate set: from which a solution is created
- a selection function: chooses best candidate to be added to the solution
- a feasibility function: determines if a candidate can be used to contribute to a solution
- an objective function: assigns a value to a solution (or partial solution)
- a solution function: indicates when we discover a complete solution

Examples:

- Traveling Salesman
  - Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city
- min coins to give change
  - pick biggest coin possible, next biggest possible, etc.
- minimum spanning tree
  - Kruskal's
  - Prim's
- optimum Huffman trees

Sources:

- `https://en.wikipedia.org/wiki/Greedy_algorithm`

Shortest Path

Dijkstras:

Implementation with PQ

- O(|E| + |V| log |V|)
- Make source current node with its distance 0
- Repeat until no elements in PQ
  - If current node is not in distance map or has a greater value than computed value place the current node in the distance map (mapped to distance)
  - Place its neighbors in the PQ with their distances to current node + current node distance
  - Dequeue min element from PQ and make current node

Bellman Ford:

psuedo java code

```
for i=1 to size(vertices)-1
   for each edge (u,v)
      if distance[u]+w < distance[v]
         distance[v]=distance[u]+w
         predecessor[v]=u
```

c++ code

```
function BellmanFord(list vertices, list edges, vertex source)
   ::distance[],predecessor[]

   // This implementation takes in a graph, represented as
   // lists of vertices and edges, and fills two arrays
   // (distance and predecessor) with shortest-path
   // (less cost/distance/metric) information

   // Step 1: initialize graph
   for each vertex v in vertices:
       if v is source then distance[v] := 0
       else distance[v] := inf
       predecessor[v] := null

   // Step 2: relax edges repeatedly
   for i from 1 to size(vertices)-1:
       for each edge (u, v) in Graph with weight w in edges:
           if distance[u] + w < distance[v]:
               distance[v] := distance[u] + w
               predecessor[v] := u
```

```
    // Step 3: check for negative-weight cycles
    for each edge (u, v) in Graph with weight w in edges:
        if distance[u] + w < distance[v]:
            error "Graph contains a negative-weight cycle"
    return distance[], predecessor[]
```

Notes

- O(|V| |E|)
- Allows negative weights

Minimum Spanning Tree

Boruvkas:

Algorithm for minimum spanning tree (smallest weight subgraph with all vertices) of a graph O(Elog V) where E=edges, v=vertices in the graph

- Find min edge for all vertices
- Connect those edges
- Loop until all connected
  - Find min edge out of all trees (connected vertices)
  - Connect those edges

O(Elog V) where E=edges, v=vertices in the graph

Notes:

- Prims
- Kruskal

Graph Contraction

Types:

- edge: two vertices connected by an edge are contracted
- star: one vertex center of stars and all vertices directly connected are contracted
- tree: disjoint tree identified and contraction performed on trees

Notes:

- Can be used to find min span tree