Classification

Classification by purpose:

- each algorithm has a goal
- kind of purposes
  - Sorting a list

Classification by implementation:

- Recursive or iterative
  - a recursive algorithm calls itself repeatedly until a certain condition matches
  - a iterative algorithm uses looping statements such as for loop, while loop or do-while loop
  - every recursive version has an iterative equivalent iterative, and vice versa
- Logical or procedural
  - an algorithm may be viewed as controlled logical deduction
  - a logic component expresses the axioms which may be used in the computation
  - a control component determines the way in which deduction is applied to the axioms
- Serial or parallel
  - in serial algorithms, computers execute one instruction of an algorithm at a time
  - parallel algorithms take advantage of computer architectures to process several instructions at once
- Deterministic or non-deterministic
  - deterministic algorithms solve the problem with a predefined process
  - non-deterministic algorithm must perform guesses of best solution at each step through the use of heuristics

Classification by design paradigm:

- Divide and conquer
- Contraction (Reduction/transform and conquer)

- Dynamic programming
- Greedy method
  - similar to dynamic programming but solutions to subproblems do not have to be known at each stage
  - a ”greedy” choice can be made of what looks the best solution for the moment
  - Kruskal
- Linear programming
- Graphs
- The probabilistic and heuristic paradigm
  - Probabilistic
    - Those that make some choices randomly
  - Genetic
    - Attempt to find solutions to problems by mimicking biological evolutionary processes
    - a cycle of random mutations yielding successive generations of ”solutions”
    - thus, they emulate reproduction and ”survival of the fittest”
  - Heuristic
    - whose general purpose is not to find an optimal solution, but an approximate solution where the time or resources to find a perfect solution are not practical

---

Sources:

- `https://www.quora.com/Which-are-the-10-algorithms-every-computer-science-student-must-imp`

Sorts

Parameters:

- inplace
- stability-maintain relative order for ”equal” keys

Accolades:

- Simplest-selection
- Efficient-MergeSort
- insertion best n average n^2 worst n^2
- selection best n^2 average n^2 worst n^2
- merge best $nlogn$ average $nlogn$ worst $nlogn$
- heap best $nlogn$ average $nlogn$ worst $nlogn$
- quick best $nlogn$ average $nlogn$ worst n^2

Classifications:

- Simple (insertion, selection)
- Efficient (Merge, Heap, quick)
- Bubble
- Distribution (bucket)
- Exchange sorts (bubble sort, quicksort)
- Selection sorts (shaker sort, heapsort)

insertion:

- take one element at a time insert into place in another list
- first element
- 2nd after or before first

- 3rd in between after or before

Selectionsort:

- look for smallest, place
- next smallest, place

Quickort:

- Pick an element, called a pivot, from the array.
- Reorder the array so that all elements with values less than the pivot come before the pivot, all elements with values greater than the pivot come after it (equal values can go either way)
- Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values

---

merge:

- Divide into two lists
- Recursively sort 2 lists
- merge two lists (add in least of both comparing them first)

---

bucketsort:

- Create buckets by range (0-9,10-19,20-29,etc)
- Separate into different buckets
- Sort buckets (using either bucketsort, quicksort, what-ever)

---

heapsort:

- Create heap of all elements
- Keep removing max (min) from heap to place into sorted array

bubblesort:

- iterate through array swapping consecutive elements if in the wrong order
- run until an interation with no swaps

sources:

- http://betterexplained.com/articles/sorting-algorithms/
- https://en.wikipedia.org/wiki/Sorting_algorithm

Recursive/Iterative

Definitions:

- Recursive
  - a function calls itself again and again till the base condition(stopping condition) is satisfied
  - common to functional programming
- Iterative
  - iterative is used to describe a situation in which a sequence of instructions can be executed multiple times
  - each time an iteration

Recursive Rules:

- each recursive call should be on a smaller instance of the same problem, that is, a smaller subproblem
- recursive calls must eventually reach a base case, which is solved without further recursion
  - Base case
  - Recursive Case
- Dynamic Programming and recursion
  - use when ever you compute a recursive input multiple times
  - memoization (caching previosly computed results, use result next time computation is needed)

Example problems:

- drawing fractals
- factorial
- towers of hanoi

Comparison:

- factorial
  - Iterative

```
def factorial(n):
    factorial = 1
    for i in range(2,n+1):
        factorial *= i
    return factorial
```

  - Recursive

```
def factorial(n):
    if (n < 2):
        return 1
    else:
        return n*factorial(n-1)
```

- Generally recursion more elegant, iteration better performance and debugability
- Iterative algorithm
  - more lines of code
  - faster
- Recursive algorithm
  - complex to implement
  - code will be elegant and easy to read
  - tracing is difficult
  - takes more time because of overheads like calling functions and registering stacks repeatedly
  - some complex problems can be solved easily and effectively in recursion

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

Back Propagation

Definitions:

Backpropagation is the central mechanism by which neural networks learn. It is the messenger telling the network whether or not the net made a mistake when it made a prediction.

Sources:

- http://neuralnetworksanddeeplearning.com/chap2.html
- https://medium.com/datathings/neural-networks-and-backpropagation-explained-in-a-simple-w
- https://medium.com/datathings/neural-networks-and-backpropagation-explained-in-a-simple-w
- https://pathmind.com/wiki/backpropagation

Convolutional (CNN)

Definitions:

Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms.

Sources:

- https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the
- http://deeplearning.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/

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`