Algorithms

- Dynamic programming – compute solutions for smaller instances of a given problem and use these solutions to construct a solution to the problem. For example: adding a cache to recursive Fibonacci calculation.
- Greedy algorithms – compute a solution in stages, making choices that are locally optimum at each step; these choices are never undone.

```go
func TopDownFibonacci(n int) int {
	fibonacci := make([]int, n+1)
	return topDownFibonacci(n, fibonacci)
}

func topDownFibonacci(n int, fibonacci []int) int {
	if fibonacci[n] > 0 {
		return fibonacci[n]
	} else {
		if n == 0 || n == 1 {
			fibonacci[n] = n
		} else {
			fibonacci[n] = topDownFibonacci(n-1, fibonacci) +
topDownFibonacci(n-2, fibonacci)
		}
		return fibonacci[n]
	}
}

func BottomUpFibonacci(n int) int {
	fibonacci := make([]int, n+1)
	if n >= 1 {
		fibonacci[0], fibonacci[1] = 0, 1
	}
	for i := 2; i <= n; i++ {
		fibonacci[i] = fibonacci[i-1] + fibonacci[i-2]
	}
	return fibonacci[n]
}
```

```go
func max(a int, b int) int {
	if a > b {
		return a
	} else {
		return b
	}
}

func knapsack() {
	const c = 10                        // capacity
	const n = 5                         // number of items
	v := [n + 1]int{0, 2, 4, 8, 10, 6} // value of items
	w := [n + 1]int{0, 1, 2, 4, 5, 3}  // weight of items
	m := [n + 1][c + 1]int{}

	// Memoization: storing precomputed values
	for i := 1; i <= n; i++ {
		for j := 0; j <= c; j++ {
			if w[i] > j {
				m[i][j] = m[i-1][j]
			} else {
				m[i][j] = max(m[i-1][j], m[i-1][j-
w[i]]+v[i])
			}
		}
	}

	for i := 0; i <= n; i++ {
		fmt.Println(m[i])
	}
	fmt.Printf("The maximum value of the %v items in %v capacity is
%v.\n", n, c, m[n][c])
}
```

Divide-and-Conquer

- Break a complex algorithm into small, often recursive parts.
- Allow better parallelization.
- To use divide-and-conquer as an algorithm design technique, we must divide the problem into two smaller subproblems, solve each of them recursively, and then meld the two partial solutions into one solution to the full problem. Whenever the merging takes less time than solving the two subproblems, we get an efficient algorithm.
- Mergesort is a classic example of divide and conquer (the merge operation is linear).

Sorting

- "Naive" sorting algorithms run in `O(n^2)` while enumerating all pairs.
- "Sophisticated" sorting algorithms run on `O(nlog(n))`.
- An important algorithm design technique is to use sorting as a basic building block, because many other problems become easy once a set of items is sorted, like: searching, finding closest pair, finding unique elements, frequency distribution, selection by order, set of numbers intersection, etc.
- *Selection Sort* – each time find the minimum item, remove it from the list, and continue to find the next minimum; takes `O(n^2)`.
- *Insertion Sort* – iterate thru `i = 2 to n`, `j = i to 1` and swap needed items; takes `O(n^2)`.
- Insertion sort is a little more efficient than selection, because the inner `j` loop uses a while, only scanning until the right place in the sorted part of the array is found for the new item. Selection sort scans all items to always find the minimum item.

- Selection and Insertion are very similar, with a difference that after `k` iterations Selection will have the `k` smallest elements in the input, and Insertion will have the arbitrary first `k` elements in the input that it processed.
- Selection Sort *writes* less to memory (Insertion writes every step because of swapping), so it may be preferable in cases where writing to memory is significantly more expensive than reading.

```
// O(n^2) average
func SelectionSort(arr []int) []int {
	arrLen := len(arr)
	for min, i := 0, 0; i < arrLen-1; i++ {
		min = i
		for j := i + 1; j < arrLen; j++ {
			if arr[j] < arr[min] {
				min = j
			}
		}
		if min != i {
			arr[i], arr[min] = arr[min], arr[i]
		}
	}
	return arr
}
```

```
// O(n^2) average
func InsertSort(arr []int) []int {
	arrLen := len(arr)
	for i := 1; i < arrLen; i++ {
		for j := i; j > 0 && arr[j] < arr[j-1]; j-- {
			arr[j], arr[j-1] = arr[j-1], arr[j]
		}
	}
	return arr
}
```

Mergesort

- A recursive approach to sorting involves partitioning the elements into two groups, sorting each of the smaller problems recursively, and then interleaving (merging) the two sorted lists to totally order the elements.
- The efficiency of mergesort depends upon how efficiently we combine the two sorted halves into a single sorted list.
- Merging on each level is done by examining the first elements in the two merged lists. The smallest element must be the head of either of the lists. Removing it, the next element must again be the head of either of the lists, and so on. So the merge operation in each level is linear.
- This yields an efficiency of `O(nlog(n))`.

- Mergesort is **great** for sorting **linked lists** because it does not access random elements directly (like heapsort or quicksort), but **DON'T** try to sort linked lists in an interview.
- When sorting arrays with mergesort an additional 3rd array buffer is ***required*** for the merging operation (can be implemented in-place tho without an additional buffer, but requires complicated buffer manipulation).
- Classic *divide-and-conquer* algorithm, the key is in the merge implementation.

```go
// O(n log n) always
func MergeSort(arr []int) []int {
	arrLen := len(arr)
	if arrLen <= 1 {
		return arr
	}

	middleIndex := arrLen / 2

	leftPart := MergeSort(arr[:middleIndex])
	rightPart := MergeSort(arr[middleIndex:])

	return merge(leftPart, rightPart)
}

func merge(left, right []int) []int {
	leftLen := len(left)
	rightLen := len(right)
	ret := make([]int, 0, leftLen+rightLen)
	for ; leftLen > 0 || rightLen > 0; leftLen, rightLen = len(left),
len(right) {
		if leftLen == 0 {
			return append(ret, right...)
		}
		if rightLen == 0 {
			return append(ret, left...)
		}

		if right[0] < left[0] {
			ret = append(ret, right[0])
			right = right[1:]
		} else {
			ret = append(ret, left[0])
			left = left[1:]
		}
	}
	return ret
}
```

Quicksort

- Can be done in-place (using swaps), doesn't require an additional buffer.

- Select a random item `p` from the items we want to sort, and split the remaining `n-1` items to two groups, those that are above `p` and below `p`. Now sort each group in itself. This leaves `p` in its exact place in the sorted array.
- Partitioning the remaining `n-1` items is linear (this step is equivalent to the "merge" part in merge sort; merge => partition).
- Total time is `O(n * h)` where `h` = height of the recursion tree (number of recursions).
- If we pick the median element as the pivot in each step, `h = log(n)`; this is the best case of quicksort.
- If we pick the left- or right-most element as the pivotal element each time (biggest or smallest value), `h = n` (worst case).
- On average quicksort will produce good pivots and have `nlog(n)` efficiency (like binary search trees insertion).
- If we are most unlucky, and select the extreme values, quicksort becomes selection sort and runs `O(n^2)`.
- For the best case to work, we need to actually select the pivot randomly, or always select a specific index and randomize the input array beforehand.
- Randomization is a powerful tool to improve algorithms with bad worst-case but good average-case complexity.
- Quicksort can be applied to real world problems – like the *Nuts and Bolts* problem (first find a match between a random nut and bolt, then split the rest into two groups: bigger and smaller. Repeat in each group).
- Experiments show that a well implemented quicksort in typically 2-3 times faster than mergesort or heapsort. The reason is the innermost loop operations are simpler. This could change on specific real world problems, because of system behavior and implementation. They have the same asymptotic behavior after all. Best way to know is to implement both and test.

```go
// O(n log n); O(n log n); O(n^2) presorted
func QuickSort(arr []int) []int {
	arrLen := len(arr)
	if arrLen <= 1 {
		return arr
	}

	// Avoid O(n^2) worst-case
	median := arr[rand.Intn(arrLen)]

	lowerPart := make([]int, 0, arrLen)
	middlePart := make([]int, 0, arrLen)
	higherPart := make([]int, 0, arrLen)

	// skip index, require value only
	for _, item := range arr {
		switch {
		case item < median:
			lowerPart = append(lowerPart, item)
		case item == median:
			middlePart = append(middlePart, item)
		case item > median:
			higherPart = append(higherPart, item)
		}
```

```
        }

        lowerPart, higherPart = QuickSort(lowerPart),
QuickSort(higherPart)

        lowerPart = append(lowerPart, middlePart...)
        lowerPart = append(lowerPart, higherPart...)
        return lowerPart
}
```

Heapsort

(See also Heap in Data Structures)

- An implementation of selection sort, but with a priority queue (implemented by a balanced binary tree) as the underlying data structure, and not a linked list.
- It's an in-place sort, meaning it uses no extra memory besides the array containing the elements to be sorted.
- The first stage of the algorithm builds a max-heap from the input array (in-place) by iterating elements from last to first and calling ***heapify*** on each (cost: `O(n)`).
- Next, the largest element of the heap is the root; we "extract" it (swap the root with the last element) ***reducing*** the size of the heap (this is important), and calling ***heapify*** again for the new root of the smaller heap.
- Continue until we're done with the entire heap.
- Time complexity is `O(nlog(n))`.

Distribution Sort

- Split a big problem into sub (ordered) "buckets" which need to be sorted themselves, but then all results can just be concatenated. An example: a phone book. Split names into buckets for the starting letter of the last name, sort each bucket, and then combine all strings to one big sorted phone book. We can further split each pile based on the second letter of the last name, reducing the problem even further.
- Bucketing is effective if we are confident that the distribution of data will be **roughly uniform** (much like hash tables).
- Performance can be terrible if data distribution is not like we thought it is.

External Sort

- Allows sorting more data then can fit into memory.
- For example, assume we have 900MB file, 100MB RAM.
- Read 100MB chunks of the data to memory and sort (quicksort, mergesort, etc.) then save to file, until all 900MB is sorted in chunks.
- Read the first 10MB of each sorted chunk to input buffers (=90MB) + allocate an output buffer (10MB) – sizes can be adjusted.
- Perform a 9-way merge (mergesort is 2-way by default) and store the result in the output buffer.
- Once the output buffer is full, write it to disk to the sorted destination file, empty it, and continue merging the input buffers.
- If any of the input buffers gets empty, fill it with the next 10MB from its chunk.

- Continue until all chunks are processed.
- Total runtime is `nlog(n)`.
- If we have a lot of data and limited RAM, we can run the whole process in two passes: split to chunks (500 files), combine 25 chunks at a time resulting in 20 larger chunks, run second merge pass to merge the 20 larger sorted chunks.

Linear Sorting Algorithms

- Counting sort – `O(n)`, stable, assuming input is integers between `0...k` and that `k = O(n)` – create a new array that first stores the number of appearances for each index, and then accumulate each of the cells to know how many total elements were before each index.
- Radix sort – use counting sort multiple times to sort on the least significant digit, then the next one, etc.
- Bucket (Bin) sort – `O(n)`, sorts `n` ***uniformly*** spread real numbers between `[0,M)`, by dividing the elements into `M/n` buckets, then sorting each bucket. Uniform distribution will yield linear time; non-uniform will be `O(nlogn)`.

Odd Even Sort

```go
// O(n^2)
func OddEvenSort(arr []int) []int {
	arrLen := len(arr)
	for isSorted := false; isSorted == false; {
		isSorted = true
		for i := 0; i < arrLen-1; i += 2 {
			if arr[i] > arr[i+1] {
				arr[i], arr[i+1] = arr[i+1], arr[i]
				isSorted = false
			}
		}
		for i := 1; i < arrLen-1; i += 2 {
			if arr[i] > arr[i+1] {
				arr[i], arr[i+1] = arr[i+1], arr[i]
				isSorted = false
			}
		}
	}
	return arr
}
```

Gnome Sort

```go
// O(n^2) -> O(n) if the list is initially almost sorted
func GnomeSort(arr []int) []int {
	arrLen := len(arr)
	for i := 1; i < arrLen; {
		if arr[i-1] > arr[i] {
			arr[i-1], arr[i] = arr[i], arr[i-1]
			if i > 1 {
```

```
                                i--
                        }
                } else {
                        i++
                }
        }
        return arr
}
```

Shell Sort

```
// O(n log n) best case -> O(n) worst-case
func ShellSort(arr []int) []int {
        arrLen := len(arr)
        for gap := int(arrLen / 2); gap > 0; gap /= 2 {
                for i := gap; i < arrLen; i++ {
                        for j := i; j >= gap && arr[j-gap] > arr[j]; j -=
gap {
                                arr[j], arr[j-gap] = arr[j-gap], arr[j]
                        }
                }
        }
        return arr
}
```

Counting Sort

```
// O(n + k) where k is the range of numbers and n is the input size
func CountingSort(arr []int) []int {
        k := getK(arr)
        counts := make([]int, k)

        arrLen := len(arr)
        for i := 0; i < arrLen; i++ {
                counts[arr[i]] += 1
        }

        for i, j := 0, 0; i < k; i++ {
                for {
                        if counts[i] > 0 {
                                counts[i] -= 1
                                arr[j] = i
                                j += 1
                        } else {
                                break
                        }
                }
        }
        return arr
}

func getK(arr []int) int {
        arrLen := len(arr)
        if arrLen == 0 {
                return 1
        }

        k := arr[0]
        for _, item := range arr {
                if item > k {
                        k = item
                }
        }
        return k + 1
}
```

Bubble Sort

```
// O(n) already sorted; O(n^2); O(n^2)
func BubbleSort(arr []int) []int {
        arrLen := len(arr)
        for i := 0; i < arrLen; i++ {
                for j := 0; j < arrLen-1; j++ {
                        if arr[j] > arr[j+1] {
                                arr[j], arr[j+1] = arr[j+1], arr[j]
                        }
                }
        }
        return arr
}
```

Cocktail Sort

```
// O(n^2)
func CocktailSort(arr []int) []int {
        arrLen := len(arr)
        for i := 0; i < arrLen/2; i++ {
                for left, right := 0, arrLen-1; left <= right; left, right
= left+1, right-1 {
                        if arr[left] > arr[left+1] {
                                arr[left], arr[left+1] = arr[left+1],
arr[left]
                        }
                        if arr[right-1] > arr[right] {
                                arr[right-1], arr[right] = arr[right],
arr[right-1]
```

```
                        }
                }
        }
        return arr
}
```

Comb Sort

```
// O(n^2 / 2^p) average, where p is the number of increments -> O(n^2)
worst-case -> O(n log n) best-case
func CombSort(arr []int) []int {
        arrLen := len(arr)
        gap := arrLen
        for {
                if gap > 1 {
                        // k = 1.3 has been suggested as an ideal shrink
factor by the authors of the original article after empirical testing on
over 200,000 random lists
                        gap = gap * 100 / 130
                }
                for i := 0; i+gap < arrLen; i++ {
                        if arr[i] > arr[i+gap] {
                                arr[i], arr[i+gap] = arr[i+gap], arr[i]
                        }
                }
                if gap == 1 {
                        break
                }
        }
        return arr
}
```

Searching

Binary Search

- Fast algorithm for searching in a **sorted** array of keys.
- To search for key `q`, we compare `q` to the middle key `S[n/2]`. If `q` appears before `S[n/2]`, it must reside in the top half of `S`; if not, it must reside in the bottom half of `S`. Repeat recursively.
- Efficiency is `O(logn)`.
- Several interesting algorithms follow from simple variants of binary search:
    - Eliminating the "equals check" in the algorithm (keeping the bigger/smaller checks) we can find the boundary of a block of identical occurrences of a search term. Repeating the search with the smaller/bigger checks swapped, we find the other boundary of the block.
    - One-sided binary search: if we don't know the size of the array, we can test repeatedly at larger intervals (`A[1]`, `A[2]`, `A[4]`, `A[8]`, `A[16]`, `...`) until we find an item larger than our search term, and then narrow in using regular binary search. This results in `2*log(p)` (where `p` is the index

we're after), regardless how large the array is. This is most useful when `p` is relatively close to our start position.

- Binary search can be used to find the roots of continuous functions, assuming that we have two points where `f(x) > 0` and `f(x) < 0` (there are better algorithms which use interpolation to find the root faster, but binary search still works well).

```go
// O(log n) always
func BinarySearch(arr []int, find int) int {
	low, high := arr[0], arr[len(arr)-1]
	if find < low || high < find {
		return -1
	}

	for low <= high {
		mid := (low + high) / 2
		switch {
		case arr[mid] < find:
			low = mid + 1
		case arr[mid] > find:
			high = mid - 1
		case arr[mid] == find:
			return mid
		}
	}
	return -1
}
```

Linear Search

```go
func LinearSearch(arr []int, find int) int {
	for i, item := range arr {
		if item == find {
			return i
		}
	}
	return -1
}
```

Randomization

- Randomize array (`O(nlogn)`) – create a new array with "priorities", which are random numbers between `1 - n^3`, then sort the original array based on new priorities array as the keys.
- Randomize array in place (`O(n)`) – swap `a[i]` with `a[rand(i, n)]`.

Selection (`k`-th smallest element)

- Sort array and return `k`-th element, `O(nlogn)`.

- Quickselect – `O(n)` expected, `O(n^2)` worst – like quicksort with the partition method, but only recurses into one of the parts (where `k` is).
- Median of medians select, `O(n)`:
    - Based on quickselect.
    - Finds an approximate median in linear time – this is the ***key*** step – which is then used as a pivot in quickselect.
    - It uses an (asymptotically) optimal approximate median-selection algorithm to build an (asymptotically) optimal general selection algorithm.
    - Can also be used as pivot strategy in with quicksort, yielding an optimal algorithm, with worst-case complexity `O(nlogn)`.
    - In practice, this algorithm is typically outperformed by instead choosing random pivots, which has average linear time for selection and average log-linear time for sorting, and avoids the overhead of computing the pivot.
    - The algorithm divides the array to groups of size 5 (the last group can be of any size <= 5) and then calculates the median of each group by sorting and selecting the middle element.
    - It then finds the median of these medians by recursively calling itself, and selects the median of medians as the pivot for partition.

Graph Algorithms

BFS Graph Traversal

- Breadth-first search usually serves to find shortest-path distances from a given source in terms of ***number of edges*** (not weight).
- Start exploring the graph from the given root (can be any vertex) and slowly progress out, while processing the oldest-encountered vertices first.
- We assign a direction to each edge, from the discoverer `(u)` to the discovered `(v)` (`u` is the parent of `v`).
- This defines (results in) a tree of the vertices starting with the root (breadth-first tree).
- The tree defines the shortest-path from the root to every other node in the tree, making this technique useful in shortest-path problems.
- Once a vertex is discovered, it is placed in a queue. Since we process these vertices in first-in, first-out order, the oldest vertices are expanded first, which are exactly those closest to the root.
- During the traversal, each vertex discovered has a parent that led to it. Because vertices are discovered in order of increasing distance from the root, the unique path from the root to each node uses the smallest number of edges on any root to node path in the graph.
- This path has to be constructed backwards, from the destination node to the root.
- BFS can have any node as the root – depends on what paths we want to find.
- BFS returns the shortest-path in terms of ***number of edges*** (for both directed and undirected graphs), not in terms of weight (see shortest-paths below).
- Traversal is `O(n + m)` where `n` = num of vertex and `m` = total num of edges.
- Applications:
    - Find shortest-path in terms of number of edges
    - Garbage collection scanning
    - Connected components
        - "Connected graph" = there is a path between ***any*** two vertices.
        - "Connected component" = set of vertices such that there is a path between every pair of vertices; there must not be a connection between two components (otherwise they would be

the same component).
        - Many problems can be reduced to finding or counting connected components (like solving a Rubik's cube – is the graph of legal configurations connected?).
        - A connected component can be found with BFS – every node found in the tree is in the same connected component. Repeat the search from any undiscovered vertex to define the next component, until all vertices have been found.
    - Vertex-colorings (testing a graph for bipartite-ness)
        - Assigning colors to vertices (as few colors as possible), such that no edge connects two vertices with the same color.
        - Such problems arise in scheduling applications (like register allocation in compilers).
        - A graph is "bipartite" if it can be colored without conflicts with only two colors.
        - Such graphs arise in many applications, like: had-sex-with graph for heterosexuals (men only have sex with women), that is male vertices are only connected to female vertices, and vice-versa.
        - The problem – find the separation between the vertices for the two colors.
        - Such a graph can be validated with BFS: start with the root, and keep coloring each new vertex the opposite color of it's parent. Then make sure that no non-discovered edge links two vertices of the same color. If no conflicts are found – problem solved. Otherwise, it isn't a bipartite graph.
        - BFS allows us to separate vertices into two groups after running this algorithm, which we can't do just from the structure of the graph.

Example

https://github.com/jiangyang5157/go-graph/blob/master/example/traversal/bfs.go

DFS Graph Traversal

- Depth-first search is often a subroutine in another algorithm.
- Starts exploring the graph from the root, but immediately expanding as far as possible (in contrast to BFS: breadth vs depth). Retraction back is only done when all neighboring vertices have already been processed.
- While BFS relied on a queue (FIFO) for new discovered vertices to be processed, DFS relies on stack (LIFO) (not really uses a stack but recursion, which functions as a stack).
- DFS classifies all edges into 2 categories: tree edges and back edges. Tree edges progress you down the tree to next vertices. Back edges take you back into some earlier part on the tree, to some ancestor.
- Start with the root, find the first child, process it, and run DFS on it recursively.
- After processing all reachable vertices, if any undiscovered vertices remain, depth-first search selects one of them as a new root and repeats from that root. The algorithm is repeated until it has discovered every vertex.
- Each vertex is timestamped (by incremented int): once when it's discovered and once when a vertex's edges have been examined.
- This defines (results in) a depth-first forest, comprising of several depth-first trees.
- Traversal is `O(n + m)` where `n` = num of vertices and `m` = total num of edges.
- Applications:
    - Topological sort – create a linear ordering of a DAG (directed acyclic graph), such that for edge `(u, v)` then `u` appears before `v` in the ordering; computed by DFS and adding each "visited" vertex into

the *front* of a linked list.
  - Finding cycles – any back edge means that we've found a cycle. If we only have tree edges then there are no cycles (to detect a back edge, when you process the edge `(x,y)` check if ☑ != y).
  - Finding articulation vertices (articulation or cut-node is a vertex that removing it disconnects a connected component, causing loss of connectivity). Finding articulation vertices by brute-force is `O(n(n + m))`.
  - Strongly connected components – in a SCC of a directed graph, every pair of vertices `u` and `v` are reachable from each other.

Example

https://github.com/jiangyang5157/go-graph/blob/master/example/traversal/dfs.go

Minimum Spanning Tree

- Convert a weighted graph to a tree reaching all nodes (*spanning*), while having the minimal weight (*minimum*).

Shortest-Paths

- Find the path with minimal edge ***weights*** between a source vertex and every other vertex in the graph.
- Similarly, BFS finds a shortest-path for an ***unweighted*** graph in terms of number of edges.
- Sub-paths of shortest-paths are also shortest-paths.
- Graphs with negative-weight cycles don't have a shortest-paths solution.
- Some algorithms assume no negative weights.
- Shortest-paths have no cycles.
- We will store the path itself in `vertex.parent` attributes, similar to BFS trees; the result of a shortest-paths algorithm is a *shortest-paths tree*: a rooted tree containing a shortest-path from the source to every vertex that is reachable.
- There may be more than one shortest-path between two vertices and more than one shortest-paths tree for a source vertex.
- Each vertex `v` maintains an attribute `v.distance` which is an upper bound on the weight of a shortest-path ("*shortest-path estimate*") from source `source` to `v`; `v.distance` is initialized to `Integer.MAX_VALUE` and `source.distance` is initialized to `0`.
- The action of *relaxing* an edge `(from, to)` tests whether we can improve the shortest-path to `to` found so far by going through `from`, and if so updating `to.parent` and `to.distance` (and the priority queue).

Dijkstra's Algorithm

- Algorithm for finding the shortest-paths between nodes in a graph, which may represent, for example, road networks.
- Assumes that edge weight is nonnegative.
- Fixes a single node as the ***source*** node and finds shortest-paths from the source to all other nodes in the graph, producing a shortest-path tree.
- The algorithm uses a min-priority queue for vertices based on their `.distance` value (shortest-path estimate).

- It also maintains a set of vertices whose final shortest-path weights from the source have already been determined.

Example

https://github.com/jiangyang5157/go-graph/blob/master/example/dijkstra/dijkstra.go

A*

- An extension of Dijkstra which achieves better time performance by using heuristics.

Backtracking

- General algorithm for finding all (or some) solutions to a problem, that incrementally builds candidates to the solution, and abandons ("backtracks") each partial candidate as soon as it determines that it cannot possibly be completed to a valid solution.
- The classic textbook example of the use of backtracking is the eight queens puzzle; another one is the knapsack problem.
- Backtracking can be applied only for problems which admit the concept of a "partial candidate solution" and a relatively quick test of whether it can possibly be completed to a valid solution.
- When it is applicable, however, backtracking is often much faster than brute force enumeration of all complete candidates, since it can eliminate a large number of candidates with a single test.

Lavenshtein Distance

- A string metric for measuring the difference between two sequences.
- The Levenshtein distance between two words is the minimum number of single-character edits (i.e. insertions, deletions or substitutions) required to change one word into the other.