Concept Notes

Design & Analysis of Algorithm

MCA III Semester

Instructor:
Nitin Deepak, Asst. Professor,
Amrapali Institute

Algorithms

INTRODUCTION

An algorithm, named after the ninth century scholar *Abu Jafar Muhammad Ibn Musu Al-Khowarizmi*, is defined as follows: Roughly speaking:

- An algorithm is a set of rules for carrying out calculation either by hand or on a machine.
- An algorithm is a finite step-by-step procedure to achieve a required result.
- An algorithm is a sequence of computational steps that transform the input into the output.
- An algorithm is a sequence of operations performed on data that have to be organized in data structures.
- An algorithm is an abstraction of a program to be executed on a physical machine (model of Computation).

The most famous algorithm in history dates well before the time of the ancient Greeks: this is the Euclid's algorithm for calculating the greatest common divisor of two integers. This theorem appeared as the solution to the Proposition II in the Book VII of Euclid's "Elements." Euclid's "Elements" consists of thirteen books, which contain a total number of 465 propositions.

ALGORITHM PERFORMANCE

Two important ways to characterize the effectiveness of an algorithm are its space complexity and time complexity. Time complexity of an algorithm concerns determining an expression of the number of steps needed as a function of the problem size. Since the step count measure is somewhat coarse, one does not aim at obtaining an exact step count. Instead, one attempts only to get asymptotic bounds on the step count. Asymptotic analysis makes use of the O (Big Oh) notation. Two other notational constructs used by computer scientists in the analysis of algorithms are Θ (Big Theta) notation and Ω (Big Omega) notation.

The performance evaluation of an algorithm is obtained by totaling the number of occurrences of each operation when running the algorithm. The performance of an algorithm is evaluated as a function of the input size n and is to be considered modulo a multiplicative constant.

Growth of Function

To characterize the time cost of algorithms, we focus on functions that map input size to (typically, worst-case) running time. (Similarly for space costs.) We are interested in precise notation for characterizing running-time differences that are likely to be significant across different platforms and different implementations of the algorithms.

- This naturally leads to an interest in the “asymptotic growth" of functions.
- We focus on how the function behaves as its input grows large.
- Asymptotic notation is a standard means for describing families of functions that share similar asymptotic behavior.
- Asymptotic notation allows us to ignore small input sizes, constant factors, lower-order terms in polynomials, and so forth.

Recurrence

A recurrence is a recursive description of a function, or in other words, a description of a function in terms of itself. Like all recursive structures, a recurrence consists of one or more base cases and one or more recursive cases. Each of these cases is an equation or inequality, with some function value f (n) on the left side. The base cases give explicit values for a (typically finite, typically small) subset of the possible values of n. The recursive cases relate the function value f (n) to function value f (k) for one or more integers k < n; typically, each recursive case applies to an infinite number of possible values of n.

For example, the following recurrence (written in two different but standard ways) describes the identity function. In both presentations, the first line is the only base case, and the second line is the only recursive case. The same function can satisfy many different recurrences; for example, both of the following recurrences also describe the identity function *f(n)=n*:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ f(n-1)+1 & \text{otherwise} \end{cases} \qquad \begin{array}{l} f(0) = 0 \\ f(n) = f(n-1)+1 \text{ for all } n > 0 \end{array}$$

In both presentations, the first line is the only base case, and the second line is the only recursive case. The same function can satisfy many different recurrences; for example, both of the following recurrences also describe the identity function:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) & \text{otherwise} \end{cases} \qquad f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2 \cdot f(n/2) & \text{if } n \text{ is even and } n > 0 \\ f(n-1)+1 & \text{if } n \text{ is odd} \end{cases}$$

Sorting Algorithms

In computer science, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

- The output is in nondecreasing order (each element is no smaller than the previous element according to the desired total order);
- The output is a permutation (reordering) of the input.

Two types of sorting:

Comparison

Comparison sorts always compares between keys to sort the whole list or keys.

Stability

Stable sorting algorithms maintain the relative order of records with equal keys. If all keys are different then this distinction is not necessary. But if there are equal keys, then a sorting algorithm is stable if whenever there are two records (let's say R and S) with the same key, and R appears before S in the original list, then R will always appear before S in the sorted list. When equal elements are indistinguishable, such as with integers, or more generally, any data where the entire element is the key, stability is not an issue.

Some of the popular sorting techniques as below:

Bubble Sort

Bubble sort is a simple sorting algorithm. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. This algorithm's average and worst case performance is O(n2), so it is rarely used to sort large, unordered, data sets. Bubble sort can be used to sort a small number of items (where its asymptotic inefficiency is not a high penalty). Bubble sort can also be used efficiently on a list of any length that is nearly sorted (that is, the elements are not significantly out of place). For example, if any number of elements are out of place by only one position (e.g. 0123546789 and 1032547698), bubble sort's exchange will get them in order on the first pass, the second pass will find all elements in order, so the sort will take only 2n time.

Selection Sort

Selection sort is an in-place comparison sort. It has O(n2) complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations.

The algorithm finds the minimum value, swaps it with the value in the first position, and repeats these steps for the remainder of the list. It does no more than n swaps, and thus is useful where swapping is very expensive.

Insertion Sort

Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly sorted lists, and often is used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one. Shell sort (see below) is a variant of insertion sort that is more efficient for larger lists.

Shell Sort

Shell sort was invented by Donald Shell in 1959. It improves upon bubble sort and insertion sort by moving out of order elements more than one position at a time. One implementation can be described as arranging the data sequence in a two-dimensional array and then sorting the columns of the array using insertion sort.

Comb Sort

Comb sort is a relatively simplistic sorting algorithm originally designed by Wlodzimierz Dobosiewicz in 1980. Later it was rediscovered and popularized by Stephen Lacey and Richard Box with a Byte Magazine article published in April 1991. Comb sort improves on bubble sort, and rivals algorithms like Quicksort. The basic idea is to eliminate turtles, or small values near the end of the list, since in a bubble sort these slow the sorting down tremendously. (Rabbits, large values around the beginning of the list, do not pose a problem in bubble sort)

Merge Sort

Merge sort takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (i.e., 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until at last two lists are merged into the final sorted list. Of the algorithms described here, this is the first that scales well to very large lists, because its worst-case running time is O(n log n). Merge sort has seen a relatively recent surge in popularity for practical implementations, being used for the standard sort routine in the programming languages Perl,[12] Python (as timsort[13]), and Java (also uses timsort as of JDK7[14]), among others. Merge sort has been used in Java at least since 2000 in JDK1.3.[15][16]

Heap Sort

Heapsort is a much more efficient version of selection sort. It also works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a data structure called a heap, a special type of binary tree. Once the data list has been made into a heap, the root node is guaranteed to be the largest (or smallest) element. When it is removed and placed at the end of the list, the heap is rearranged so the largest element remaining moves to the root. Using the heap, finding the next largest element takes O(log n) time, instead of O(n) for a linear scan as in simple selection sort. This allows Heapsort to run in O(n log n) time, and this is also the worst case complexity.

Quick Sort

Quicksort is a divide and conquer algorithm which relies on a partition operation: to partition an array an element called a pivot is selected. All elements smaller than the pivot are moved before it and all greater elements are moved after it. This can be done efficiently in linear time and in-place. The lesser and greater sublists are then recursively sorted. Efficient implementations of quicksort (with in-place partitioning) are typically unstable sorts and somewhat complex, but are among the fastest sorting algorithms in practice. Together with its modest O(log n) space usage, quicksort is one of the most popular sorting algorithms and is available in many standard programming libraries. The most complex issue in quicksort is choosing a good pivot element; consistently poor choices of pivots can result in drastically slower $O(n^2)$ performance, if at each step the median is chosen as the pivot then the algorithm works in O(n log n). Finding the median however, is an O(n) operation on unsorted lists and therefore exacts its own penalty with sorting.

Counting Sort

Counting sort is applicable when each input is known to belong to a particular set, S, of possibilities. The algorithm runs in O(|S| + n) time and O(|S|) memory where n is the length of the input. It works by creating an integer array of size |S| and using the ith bin to count the occurrences of the ith member of S in the input. Each input is then counted by incrementing the value of its corresponding bin. Afterward, the counting array is looped through to arrange all of the inputs in order. This sorting algorithm cannot often be used because S needs to be reasonably small for it to be efficient, but the algorithm is extremely fast and demonstrates great asymptotic behavior as n increases. It also can be modified to provide stable behavior.

Radix Sort

Radix sort is an algorithm that sorts numbers by processing individual digits. n numbers consisting of k digits each are sorted in O(n · k) time. Radix sort can process digits of each number either starting from the least significant digit (LSD) or starting from the most significant digit (MSD). The LSD algorithm first sorts the list by the least significant digit while preserving their relative order using a stable sort. Then it sorts them by the next digit, and so on from the least significant to the most significant, ending up with a sorted list. While the LSD radix sort requires the use of a stable sort, the MSD radix sort algorithm does not (unless stable sorting is desired). In-place MSD radix sort is not stable. It is common for the counting sort algorithm to be used internally by the radix sort. Hybrid sorting approach, such as using insertion sort for small bins improves performance of radix sort significantly.

Bucket Sort

Bucket sort is a divide and conquer sorting algorithm that generalizes Counting sort by partitioning an array into a finite number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. A variation of this method called the single buffered count sort is faster than quicksort.[citation needed]

Due to the fact that bucket sort must use a limited number of buckets it is best suited to be used on data sets of a limited scope. Bucket sort would be unsuitable for data such as social security numbers - which have a lot of variation.

Distribution Sort

Distribution sort refers to any sorting algorithm where data are distributed from their input to multiple intermediate structures which are then gathered and placed on the output. See *Bucket sort, Flashsort.*

Timsort

Timsort finds runs in the data, creates runs with insertion sort if necessary, and then uses merge sort to create the final sorted list. It has the same complexity (O(nlogn)) in the average and worst cases, but with pre-sorted data it goes down to O(n).

Data Structure

In computer science, a data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.

Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, B-trees are particularly well-suited for implementation of databases, while compiler implementations usually use hash tables to look up identifiers.

Data structures are used in almost every program or software system. Data structures provide a means to manage huge amounts of data efficiently, such as large databases and internet indexing services. Usually, efficient data structures are a key to designing efficient algorithms. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design.

Abstract data types

- Container
- Map/Associative array/Dictionary
- Multimap
- List
- Set
- Multiset
- Priority queue
- Queue
- Deque
- Stack
- String
- Tree
- Graph

Advanced Data Structure

Red-Black Tree

A red-black tree is a binary search tree with one extra attribute for each node: the colour, which is either red or black. We also need to keep track of the parent of each node, so that a red-black tree's node structure would be:

```
struct t_red_black_node {
    enum { red, black } colour;
    void *item;
    struct t_red_black_node *left, *right, *parent;
    }
```

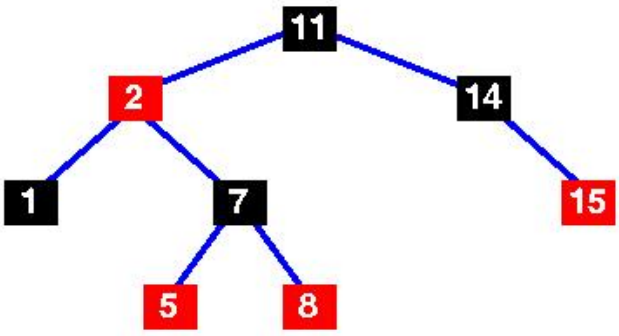
For the purpose of this discussion, the NULL nodes which terminate the tree are considered to be the leaves and are colored black.

Definition of Red-Black Tree

A red-black tree is a binary search tree which has the following red-black properties:

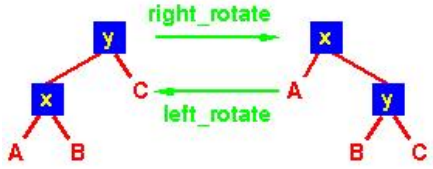
1. Every node is either red or black.
2. Every leaf (NULL) is black.
3. If a node is red, then both its children are black.
    a. Implies that on any path from the root to a leaf, red nodes must not be adjacent. However, any number of black nodes may appear in a sequence.
4. Every simple path from a node to a descendant leaf contains the same number of black nodes.

A basic Red-Black Tree



Rotations

Rotation is a local operation in a search tree that preserves in-order traversal key ordering. Note that in both trees, in-order traversal yields:



Insertion and *deletion* in Red-Black tree are the techniques to be followed during the study.

B-Trees

Tree structures support various basic dynamic set operations including Search, Predecessor, Successor, Minimum, Maximum, Insert, and Delete in time proportional to the height of the tree. Ideally, a tree will be balanced and the height will be log n where n is the number of nodes in the tree. To ensure that the height of the tree is as small as possible and therefore provide the best running time, a balanced tree structure like a red-black tree, AVL tree, or b-tree must be used.

When working with large sets of data, it is often not possible or desirable to maintain the entire structure in primary storage (RAM). Instead, a relatively small portion of the data structure is maintained in primary storage, and additional data is read from secondary storage as needed. Unfortunately, a magnetic disk, the most common form of secondary storage, is significantly slower than random access memory (RAM). In fact, the system often spends more time retrieving data than actually processing data.

B-trees are balanced trees that are optimized for situations when part or all of the tree must be maintained in secondary storage such as a magnetic disk. Since disk accesses are expensive (time consuming) operations, a b-tree tries to minimize the number of disk accesses. For example, a b-tree with a height of 2 and a branching factor of 1001 can store over one billion keys but requires at most two disk accesses to search for any node.

Binomial Heap

Bk is a binomial tree of degree k, consisting of a root and sub-trees B0, B1, B2, ...B(K-1) in order (either right to left or left to right), where each subscript less than k appears exactly once. *(and, remember, binomial trees are general trees, NOT k-ary trees)*

A binomial heap is a forest of binomial trees, where at most one of each B0, B1,...,B(k) appears in the forest, and the nodes in the binomial trees satisfy the partial order property associated with a min (max) heap: the value in a node is less than or equal to (greater than or equal to) the values in its child nodes.

Some operations on binomial heap like, find-minimum, extract-minimum, insertion, deletion, union etc.

Fibonacci Heap

A Fibonacci heap is essentially a forest of binomial heaps, except that the Fibonacci heap does not restrict the degrees of the binomial trees making up the forest.

That is, an f-heap having k roots in the forest is not required to have exactly one of each of B0, B1, B2..B(K-1). An f-heap can have any number of Bi's present in any order, except after consolidation, when it can have at most 1 of a given Bi in the forest. However, not all Bi need to be represented.

Some operations on Fibonacci Heap like find-minimum, extract-minimum, insertion, deletion, consolidation etc.

Data Structure for Disjoint Sets

A disjoint-set data structure is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. A union-find algorithm is an algorithm that performs two useful operations on such a data structure:

- Find Set: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.
- Union: Join two subsets into a single subset.

Because it supports these two operations, a disjoint-set data structure is sometimes called a union-find data structure or merge-find set. The other important operation, MakeSet, which makes a set containing only a given element (a singleton), is generally trivial. With these three operations, many practical partitioning problems can be solved.

In order to define these operations more precisely, some way of representing the sets is needed. One common approach is to select a fixed element of each set, called its representative, to represent the set as a whole. Then, Find(x) returns the representative of the set that x belongs to, and Union takes two set representatives as its arguments.

Dynamic Programming

In mathematics and computer science, dynamic programming is a method for solving complex problems by breaking them down into simpler sub-problems. It is applicable to problems exhibiting the properties of overlapping sub-problems which are only slightly smaller and optimal substructure. When applicable, the method takes far less time than naive methods.

The key idea behind dynamic programming is quite simple. In general, to solve a given problem, we need to solve different parts of the problem (sub-problems), and then combine the solutions of the sub-problems to reach an overall solution. Often, many of these sub-problems are really the same. The dynamic programming approach seeks to solve each sub-problem only once, thus reducing the number of computations: once the solution to a given sub-problem has been computed, it is stored or "memorized": the next time the same solution is needed, it is simply looked up. This approach is especially useful when the number of repeating sub-problems grows exponentially as a function of the size of the input.

Greedy Algorithms

A greedy algorithm repeatedly executes a procedure which tries to maximize the return based on examining local conditions, with the hope that the outcome will lead to a desired outcome for the global problem. Greedy algorithms do not always yield a genuinely optimal solution.

Greedy approximation algorithms have been frequently used to obtain sparse solutions to learning problems.

For example, all known greedy algorithms for the graph coloring problem and all other NP-complete problems do not consistently find optimum solutions. Nevertheless, they are useful because they are quick to think up and often give good approximations to the optimum.

Back-Tracking

Backtracking is a general algorithmic technique that considers searching every possible combination in order to solve an optimization problem. Backtracking is also known as depth-first search or branch and bound. By inserting more knowledge of the problem, the search tree can be pruned to avoid considering cases that don't look promising. While backtracking is useful for hard problems to which we do not know more efficient solutions, it is a poor solution for the everyday problems that other techniques are much better at solving.

However, dynamic programming and greedy algorithms can be thought of as optimizations to backtracking, so the general technique behind backtracking is useful for understanding these more advanced concepts. Learning and understanding backtracking techniques first provides a good stepping stone to these more advanced techniques because you won't have to learn several new concepts all at once.

Backtracking Methodology:

- View picking a solution as a sequence of choices
- For each choice, consider every option recursively
- Return the best solution found

Branch and Bound

Branch and bound (BB or B&B) is a general algorithm for finding optimal solutions of various optimization problems, especially in discrete and combinatorial optimization. It consists of a systematic enumeration of all candidate solutions, where large subsets of fruitless candidates are discarded en masse, by using upper and lower estimated bounds of the quantity being optimized.

Amortized Analysis

Imagine for example a stack. We have the following operations on the stack:

- Pop(S) pops the top element of the stack and returns it.
- Push(S,x) pushed element x on the stack.
- MultiPop(S,k) returns at most the top k elements from the stack (it calls Pop k times).

Obviously the operations Pop and Push have worst case time *O (1)*. However the operation MultiPop can be linear in the stack size. So if we assume that at most n objects are on the stack a multipop operation can have worst case cost of *O (n)*. Hence in the worst case a series of n stack operations is bounded by $O(n^2)$.
Amortized analysis is a tool for analyzing algorithms that perform a sequence of similar operations. It can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a single operation within the sequence might be expensive. Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees the average performance of each operation in the worst case.

There are three most common techniques used in amortized analysis - **Aggregate Analysis**, **Accounting Method** (also known as Taxation Method) and **Potential Method**.

Graph

A graph is a collection (nonempty set) of vertices and edges.

- **Vertices**: can have names and properties.
- **Edges**: connect two vertices, can be labeled, and can be directed.
- **Adjacent vertices**: if there is an edge between them.

Some terms related to graph theory:

Directed graphs and undirected graphs: In *undirected graphs* the edges are symmetrical, e.g. if A and B are vertices, A B and B A are one and the same edge. Graph1 above is undirected.

In *directed graphs* the edges are oriented; they have a beginning and an end. Thus A B and B A are different edges. Sometimes the edges of a directed graph are called arcs.

Paths: A path is a list of vertices in which successive vertices are connected by edges.

Simple path: No vertex is repeated.

Cycles: A cycle is a simple path with distinct edges, where the first vertex is equal to the last.

Loop: An edge that connects the vertex with itself.

Connected graphs: Connected graph: There is a path between each two vertices.

Trees: A tree is an undirected graph with no cycles and a vertex chosen to be the root of the tree.

A spanning tree of a graph: A spanning tree of an undirected graph is a sub-graph that contains all the vertices, and no cycles. If we add any edge to the spanning tree, it forms a cycle, and the tree becomes a graph.

Complete graph: Graphs with all edges present – each vertex is connected to all other vertices, are called complete graphs.

Weighted graphs: Weights are assigned to each edge (e.g. distances in a road map).

Networks: directed weighted graphs.

Graph representation: **Adjacency matrix** and **Adjacency list structure**.

Approximation Algorithms

From now on we will use opt(I) to denote the value of an optimal solution to the problem under consideration for input I. For instance, when we study TSP then opt(P) will denote the length of a shortest tour on a point set P, and when we study the independent-set problem then opt(G) will denote the maximum size of any independent set of the input graph G. When no confusion can arise we will sometimes simply write opt instead of opt(I).

A minimization problem is a problem where we want to find a solution with minimum value; TSP is an example of a minimization problem. An algorithm for a minimization problem is called a p-approximation algorithm, for some $p > 1$, if the algorithm produces for any input I a solution whose value is at most $p \cdot \text{opt}(I)$. A maximization problem is a problem where we want to find a solution with maximum value; independent set is an example of a maximization problem. An algorithm for a maximization problem is called a p-approximation algorithm, for some $p < 1$, if the algorithm produces for any input I a solution whose value is at least $p \cdot \text{opt}(I)$. The factor p is called the approximation factor (or: approximation ratio) of the algorithm.

String Pattern Matching

In computer science, pattern matching is the act of checking some sequence of tokens for the presence of the constituents of some pattern. In contrast to pattern recognition, the match usually has to be exact. The patterns generally have the form of either sequences or tree structures. Uses of pattern matching include outputting the locations (if any) of a pattern within a token sequence, to output some component of the matched pattern, and to substitute the matching pattern with some other token sequence (i.e., search and replace).

Sequence patterns (e.g., a text string) are often described using regular expressions and matched using techniques such as backtracking.

NP-Hard & NP-Completeness

A decision problem is in P if there is a known polynomial-time algorithm to get that answer. A decision problem is in NP if there is a known polynomial-time algorithm for a non-deterministic machine to get the answer.

Problems known to be in P are trivially in NP — the nondeterministic machine just never troubles itself to fork another process, and acts just like a deterministic one. There are problems that are known to be neither in P nor NP; a simple example is to enumerate all the bit vectors of length n. No matter what, that takes 2n steps.

(Strictly, a decision problem is in NP if a nondeterministic machine can arrive at an answer in poly-time, and a deterministic machine can verify that the solution is correct in poly time.)

But there are some problems which are known to be in NP for which no poly-time deterministic algorithm is known; in other words, we know they’re in NP, but don’t know if they’re in P. The traditional example is the decision-problem version of the Traveling Salesman Problem (decision-TSP): given the cities and distances, is there a route that covers all the cities, returning

to the starting point, in less than x distance? It's easy in a nondeterministic machine, because every time the nondeterministic traveling salesman comes to a fork in the road, he takes it: his clones head on to the next city they haven't visited, and at the end they compare notes and see if any of the clones took less than x distance.

It's not known whether decision-TSP is in P: there's no known poly-time solution, but there's no proof such a solution doesn't exist.

Now, one more concept: given decision problems P and Q, if an algorithm can transform a solution for P into a solution for Q in polynomial time, it's said that Q is poly-time reducible (or just reducible) to P.

A problem is NP-complete if you can prove that (1) it's in NP, and (2) show that it's poly-time reducible to a problem already known to be NP-complete.

So really, what it says is that if anyone ever finds a poly-time solution to one NP-complete problem, they've automatically got one for all the NP-complete problems; that will also mean that P=NP.

A problem is NP-hard if and only if it's "at least as" hard as an NP-complete problem. The more conventional Traveling Salesman Problem of finding the shortest route is NP-hard, not strictly NP-complete.