CS 2223 Algorithms

Christopher Myers

June 13, 2020

Contents

1 Introduction

Algorithms are computational procedures with the following properties:

- Each step is precisely stated
- Algorithms always terminate in a finite number of steps
- There is well-defined input and output

This class will focus on deterministic algorithms, which always produce the same output given the same input. Algorithms may be considered correct if, under all valid inputs, they produce the correct output. Algorithms can also be evaluated based on efficiency, since one problem can have many different possible solutions. Some algorithms may be faster than others, while others may use less CPU power, memory, or storage space to run.

2 Proving Correctness

An algorithm's correctness can be determined mathematically — testing it is not enough. Usually this is done as proof by induction, but proof by contradiction is also possible.

To facilitate proof by induction, loop invariants are used. These are statements of conductions that are true immediately before and after each iteration of a loop. Three things must be shown: initialization, maintenance (something true just before the next step), and termination (condition required for loop to end).

3 Evaluating Efficiency

Big-O notation is a form of notation for the efficiency of an algorithm; the function inside shows approximately how quickly the algorithm's processing time increases as the input size increases. Formally, $O(g(n))$ is defined to be the following set of functions: $O(g(n)) = f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

In practice, Big O notation gives an upper bound and a worst case for a given algorithm. While it is technically correct to say that $n^2 = O(n^7)$, it is more appropriate to use a tight upper bound and say that $n^2 = O(n^2)$. When evaluating expressions (e.g. $n! + n^2$), only the dominant term is preserved; here, that would mean that $n! + n^2 = O(n^2)$.

Example Show that $n^2 + 3n^3 = \Theta(n^3)$. In other words, show that $\Theta(n^3)$ asymptotically upper-bounds and lower-bounds $n^2 + 3n^3$.

This means proving two things: prove that $n^2 + 3n^3 \exists O(n^3)$ and $n^2 + 3n^3 \exists \Omega(n^3)$. So, find c and n_0 such that $n^2 + 3n^3 \leq cn^3$ where $n \geq n_0$. Here, choose $c = 4$ and $n_0 = 0$. This makes $n^2 + 3n^2 \leq 4n^3$, so the first part is proven.

For the second (Ω), find c and n_0 such that $n^2 + 3n^3 \geq cn^3$, and once again $n_0 \geq 0$. Choose $c = 3$ and $n_0 = 0$, and find that $n^2 + 3n^2 \geq 3n^3$. This proves that $n^2 + 3n^3 = \Theta(n^3)$.

3.1 Master Method

The Master method can be used to evaluate algorithms that use divide-and-conquer recursion, specifically with the form of $T(n) = aT(\frac{n}{b}) + f(n)$ where $a \geq 1, b > 1$, and $f(n) > 0$. Basically, compare $n^{\log_b a}$ to $f(n)$:

- Case 1, cost is dominated by leaf work. Use $N^{\log_b a}$.
- Case 2, cost is the same for leaves and root. Use $N^{\log_b a} \log n$
- Case 3, cost is dominated by root use. Use $F(n)$ (the last part of a recursion equation).

Example Solve the recursion equation $T(n) = 2(\frac{n}{2}) + 3n, T(1) = 1$.

Use back substitution:

$$T(n/2) = 2T(n/4) + 3(n/2)$$
$$T(n/4) = 2T(n/8) + 3(n/4)$$

Substitute the first result into the original equation:

$$\begin{aligned} T(n) &= 2[2T(n/4) + 3(n/2)] + 3n \\ &= 4T(n/4) + 6n \\ T(n) &= 4[2T(n/8) + 3(n/4)] + 6n \\ &= 8t(n/2) + 9n \end{aligned}$$

If you look at the pattern, the term in front of T is 2^k, the term inside T is $\frac{n}{2^k}$, and the final term added on is $3kn$. That makes the formula $2^k T(\frac{n}{2^k}) + 3kn$. Since $k = \log n$ and $1 = n/2^k$, solving a bit yields $2^{\log n} T(1) + 3n \log n$. Simplifying yields $n + 3n \log n$, for a final result of $n \log n$.

Alternatively, use the Master method. This formula follows the second case (n vs. $\log n$), so the result is $n \log n$.

4 Dynamic Programming

Dynamic programming, better called dynamic *planning*, is a method used to make tough-to-solve problems easier. The jist of dynamic programming is to save solutions to past components of a larger problem into one table, which can then be used to solve more of those same components, all the way up until the final solution is found. A good example of this in action in Dijkstra's algorithm — the algorithm works by solving for the most efficient path up to a point, then builds off that progress. Instead of checking each unique path (which would take an absurd amount of time), the algorithm can find the shortest path much quicker. Another good example would be to solve for numbers in the Fibonacci series by saving the results of the last calculation, rather than just redoing the whole set of series calculations for each element.

4.1 Coin-row problem

There is a row of n coins whose values $c_1, c_2, \ldots c_n$ are some positive integers that are not necessarily distinct. The goal is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in initial row can be picked up. E.g. for 5121062, the best selection is to pick the 5, the 10, and the rightmost 2 (for a total of 17).

Let $F(n)$ be the maximum amount that can be picked up from the row of n coins. Then $F(n) = \max(c_n + F(n-2), F(n-1))$ for $n > 1$, where $F(0) = 0, F(1) = c_1$. Using this method, the coin row problem can be solved in $\Theta(n)$ time and space.

4.2 Change making problem

The challenge here is to give change for an amount n using the minimum possible number of coins. Let $F(n)$ be the minimum number of coins whose values add up to n, and let $F(0) = 0$ (no change). The amount n can only be obtained by adding one coin of denomination d_j to the amount $n - d_j$ for $j = 1, 2 \ldots m$. So, $F(n) = \min(f(n - d_j), J : n = d_j) + 1$ for $n > 0$.

4.3 Coin collecting by robot

Several coins are placed in cells of an $n \times m$ board. A robot, located in the upper left cell of the board, collects the coins. Each iteration it can move by one square right or down. For a given board, how many coins can the robot collect?

Here, $F(i,j) = \max(F(i-1,j), F(i,j-1)) + C_{ij}$, (i.e. moving left or moving up), for $1 \leq i \leq n$ and $1 \leq j \leq m$, where $C_{ij} = 1$ if there is a coin in cell (i,j), but 0 otherwise. The initial conditions are $F(0,j) = 0$ for $1 \leq j \leq m$ and $F(i,0) = 0$ for $1 \leq i \leq n$. Following this method will you the largest value possible in the bottom right square; backtracking using the board of filled values will show you the best path for the robot to take.

4.4 Knapsack Problem

This isn't just a problem, it's an NP-Complete problem. So far, there is no known way to efficiently solve the problem (i.e. solve it in polynomial time or less). So far it takes exponential time or greater to solve the problem, even using fun techniques like dynamic programming. The problem itself is simple: if you have a set of items with integer weight and assigned value to each, and a space of finite size to put them, find the most valuable subset.

To solve using dynamic programming, build a grid with the capacity by weight on the horizontal axis the max index of item to select. Define a function as:

$$F(i,j) = \max(F(i-1,j), v_i + F(i-1, J - w_i))$$

If $J - w_i \geq 0$, the item fits. If $J - w_i < 0$, the item doesn't fit. What this ends up building is a grid of increasing values, with the largest possible value stored in the bottom right.

4.5 Longest Common Subsequence

LCS is an algorithm that, given two input strings, will find the longest substring that the two share.[1] In the event of equal strings, the result will be the entire string; in the case of completely different strings, the result will be an empty string.

Now for the actual algorithm. The inputs are X and Y, where each is is a string of characters. Assume that $C[i,j]$ is a matrix and that $C[i,j]$ is the LCS for the first i positions in X with the first j positions in Y. So, $C[i,j] = LCS(< x_1, x_2, x_3 \ldots x_i >, LCS < y_1, y_2, y_3, \ldots y_j >)$. In code:

```
def LCS-Length(X,Y):
    m = X.length
    n = Y.length
    let b[1..m, 1..n] and c[0..m, 0..n] be new arrays
    for i in range(1, m+1):
        c[i,0]=0
    for i in range(0, n+1):
        c[0,j]=0
    for i in range(1, m+1):
        for j in range(1, j+1)
            if x[i] == y[j]:
                c[i,j] = c[i-1, j-1]+1
                b[i,j] = ``topleft''
            elif c[i-1,j] $\geq$ c[i,j-1]:
                c[i,j]=c[i-1,j]
                b[i,j]= ``up''
            else c[i,j]=c[i,j-1]:
                b[i,j]= ``left''
    return c and b
```

[1] Note that it is acceptable for other characters to be inserted into the middle of a substring. . . ?

4.6 Edit Distance

The edit distance algorithm is an algorithm to compute the smallest "distance" between two strings S1 and S2 (of sizes n and m, respectively). Here, "distance" is given by the number of edit operations it takes to convert string S1 into string S2. Operations are defined as insert, delete, and align. For example, if you have S1="TCGACGTGCA" and S2="TGACGTGC", you could create a set of three operations to perform the transformation.

The algorithm to compute this is similar to the LCS algorithm. A matrix is assembled with S1 on the left and S2 on the top, both offset from the top left by one. 0 is placed in the top left corner. Costs are given in multipliers of i, d, and a for insert, delete, and align, respectively. The uppermost row will look like $0, i, 2i, 3i, \ldots ni$<++> while the leftmost column will look like $0, 1d, 2d, 3d, \ldots mi$. From here on out, assume that $i = 1$, $d = 1$, and a costs 0 or 1 if aligned characters are the same or different, respectively.

Each cell then contains the smallest cost for converting S1[1..i] to S2[1..j]. The cost at a given point is given by the minimum of M[i-1,j-1]+cost of aligning S1[i] to S2[j], M[i-1,j]+cost of deleting S1[i], and M[i,j-1]+cost of inserting S2[j] into S1. Here, have some code:

5 Greedy Algorithms

Greedy algorithms work using the "Greedy Grab" technique, based around the idea of solving as much of the problem as possible for a given step. On each step the choice made must be:

- Feasible — the choice satisfies the problem's constraints
- Locally optimal — best local choice among all feasible choices available on that step
- Irrevocable — the choice cannot be altered as part of later steps

An example of a greedy algorithm is a solution to the change making problem, one that most people use for such problems. Start by grabbing the largest coin that fits, then re-run the same choice using the new (reduced) remaining value. While greedy algorithms can produce an output quickly, they do not necessarily provide an *optimal* output. For example, a greedy solution for the knapsack problem will definitely find you an answer quickly (the max value that can be fit into the knapsack), but that answer might not be the highest possible answer to the problem.

6 Graphs

Graphs are data structures that connect a set of objects to form a kind of network. Objects in the graph are called "nodes" or "vertices", connections between them are called edges. Undirected graphs have connections that go both ways, while directed graphs don't have bidirectional connections. Weighted graphs have weights along their edges, i.e. a "cost" associated with traversing each edge. Graphs can be qualitatively said to be dense or spare based on how many edges they have. Applications of graphs include transportation, communication, networking, biology (food webs), software systems, scheduling, circuits, and social problems.

Programmatically, there are different ways to represent graphs. Graphs can be represented as an adjacency matrix, where connections between nodes are represented using 1s and 0s. Another option is an adjacency list, where a list of connections between nodes is stored. Adjacency matrices may be faster to access, but for undirected graphs they use twice as much memory as they need to. To save space, adjacency lists can be used instead, but they can't be accessed as quickly.

Anyways, degree of a node is the amount of connections it has. The in-degree is the number of edges entering the node, while the out-degree is the number of edges leaving the node. For undirected graphs these two values are equal. Directed graphs can contain cycles, loops of nodes strung together. If a directed graph does not contain a cycle, it is called a directed acyclic graph, or a DAG for short.

6.1 Breadth-First-Search (BFS)

BFS works by starting from a node and checking all its neighbors first, then checking all the neighbors of those neighbors, and so on until the target has been found. It is so called because it searches the breadth of what's available to it first, only expanding in depth when it needs to. Generally, this will lead to short paths.

6.2 Depth First Search

Depth first search (DFS) works by traversing the graph and going deeper whenever possible. DFS relies on a stack as its underlying data structure, and so can be implemented using recursion. Once DFS has bottomed out and can explore no further (that is, without exploring nodes it's already explored), it goes back up the stack until it can explore downward again. The time complexity of this ends up being $O(|V| + |E|)$ where V and E are the sets of vertices and edges, respectively.

While DFS is executing, edges can be labeled as:

- Forward edge — an edge that leads to an already-found descendent
- Backward edge — an edge that leads to an already-found ancestor. If these are found, the graph contains a cycle; otherwise, there are no cycles.
- Cross-edge — an edge that links between trees in a forest

6.3 Minimum Spanning Tree

The Muddy City Problem is a classic graph problem where the mayor of a city wants to create a path between every house (it gets really muddy when it rains), but in order to save money the path must be as short as possible. This is accomplished by finding a minimum spanning tree. In a graph context, this means finding the shortest possible (judged based on the weights on the edges) path between all the houses. There are numerous examples of applications for these, aside from the muddy city problem. An internet service provider laying cables down for connecting residences to their network might want to find a minimum spanning tree for their service area, for instance, in order to minimize the amount of cables they have to lay.

Kruskal's Algorithm One way of finding minimum spanning trees (MSTs) begins with sorting all the edges in ascending order of their weights. The algorithm then begins adding to the work-in-progress solution by picking the lowest weight edges, avoiding cycles wherever they might be formed. This will form a forest of trees that will gradually be connected together as more edges are added. Once only one tree remains, the algorithm is complete and a minimum spanning tree has been found. Here, have some pseudocode:

```
def kruskal(G):
for v in G.V:
    make-set(v)
for (u, v) in sorted(weight(u,v)):
    if find-set(u) != find-set(v):
        A = A u {(u,v)}
        union(u,v)
return A
```

Note that because it always tries to find a local optimum solution, among other properties, Kruskal's algorithm is classified as a greedy algorithm.

Prim's Algorithm Prim's algorithm maintains two groups of nodes — the old graph and the new graph. The new graph starts with one node, but when the algorithm completes it will be the minimum spanning tree. At each step, a node is selected from the old graph and added to the new graph; the node selected is the one whose connecting edge has the smallest weight to any node in the new graph (this is a greedy

choice). It is possible for Kruskal's and Prim's algorithm to generate two different trees, but the sums of all the edge weights in the trees will still be the same.

Pseudocode:

```
def Prim(G,w,r):
    for u in V:
        key[u] = $\infty$
        color[u]=W
    key[r]=0
    pred[r]=NIL
    Q = PriorityQueue(V)
    while len(Q) > 0:
        u = Q.min()
        for v in adj[u]:
        if color[v] == W and w[u,v] < key[v]:
            key[v] = w[u,v]
            Q.decreaseKey(v, key[v])
            pred[v] = u
        color[u] = B
```

This code doesn't actually make much sense, but whatever. If you use a min heap for the priority queue, Prim's algorithm runs in $O(E \log V)$.

6.4 Traveling Salesman Problem

The travelling salesman problem has a basic premise. A salesman wants to travel to all of the different cities on a list, but wants to take the shortest possible path between them, then go back to the original city. In essence: find the shortest path that loops through all the vertexes in a graph.

7 Computational Complexity

There is a class of problem called P (tractable problems) that can be solved in polynomial time. Another major category is NP problems (intractable), ones that cannot be solved in polynomial time... not yet, anyways. It's not known for sure whether these *can* be solved in polynomial time or not, just that nobody has managed to yet. Examples include the knapsack problem and the travelling salesman problem.

8 Algorithms

8.1 Sorting Algorithms

Sorting algorithms are some of the most fundamental algorithms in computer science. Given an array of N integers, a sorting algorithm will rearrange them such that they are in increasing or decreasing order.

8.1.1 Insertion Sort

Insertion sort is a sorting algorithm that works in a way analogous to the way a person might sort a hand of cards. It's effectively a brute force sorting algorithm, and runs in $O(n^2)$ time. To operate, insertion sort loops over the array, exchanging the next element with larger elements to its left, one by one, manually placing items in the appropriate location. Insertion sort is an in-place sorting algorithm, so it requires no more memory to run than the size of the original array. Working python code for it is below:

```
    def insertion_sort(a):
        for j in range(1, len(a)): # Runs n-1 times
            key = a[j]
            i = j - 1
            while i >= 0 and a[i] > key: # Runs at most n-1 times
```

```
            a[i+1] = a[i] # Swap previous number into next position as needed
            i = i-1
        a[i+1] = key # Insert number at appropriate location
    return a
```

Based on the comments above — particularly the nested loops — it should be obvious why this runs in n^2 time. The best est case scenario for this algorithm is a list that is already sorted, while worst case is that the list is sorted in reverse order.

To prove the correctness of this algorithm, take the loop invariant of the outermost loop. The loop initializes on the second element. For maintenance, each iteration expands the subarray by 1 but keeps the sorted propery. An element gets inserted into the array only when it is greater than the element to its left, where all elements on the left are already sorted. Termination is reached when the last element in the array is processed. The subarray has then expanded to cover the entire array and the whole thing is completely sorted.

8.1.2 Bubble Sort

Bubble sort is another sorting algorithm that relies on "bubbling up" the largest value in the list to the end. Bubblesort looks at two numbers at once, swapping the left value with the right value if the one on the left is bigger than the one on the right. This is repeated over an iteration through the whole array. When the comparison box (thingy... don't know what else to call this) reaches the end, the process restarts and the whole thing is repeated over and over again until the array is sorted. An optimization can be made to keep track of the number of sorted values at the end of the array, preventing the inner loop from iterating over pairs of numbers that have already been sorted.

```
def bubblesort(l):
    for i in range(0, l):
        for j in range(0, l-1):
            if l[j] > l[j+1]:
                temp = l[j]
                l[j] = l[j+1]
                l[j+1] = temp
```

In worst-case scenario, bubblesort runs in $O(n^2)$ time, but with modifications made to detect a lack of swaps, the best case can be reduced from $O(n^2)$ to $O(n)$.

8.1.3 Merge Sort

```
def MergeSort(A, l, r):
    if l < r:
        m = (l+r)/2
        MergeSort(A,l,m)
        MergeSort(A,m+1,r)
        Merge(A,l,m,r)
```

To figure this out mathemagically, use a recursion equation. Mergesort is very much recursive, so here $T(n) = c_1$ if $n = 1$, but $T(n) = 2T(n/2) + c_2 n$ if $n > 1, n = 2^k$ (assume that n is a power of 2). Solving this yields that merge sort runs in $n \log n$ time (a significant improvement over the other sorting algorithms). Even in best case, however, mergesort still runs in $n \log n$ time.

More generically, the recurrence equation is given by $T(n) = aT(\frac{n}{b}) + f(n)$, where a is the number of problems being solved ($a \geq 1$), B is the size of the problem ($b > 1$), and $f(n)$ is the time spent dividing. Here's an example of solving for merge sort using backwards substitution:

$$T(n) = 2T(\frac{n}{2}) + c_2 n$$
$$T(\frac{n}{2}) = 2T(n/4) + c_2 \frac{n}{2}$$
$$T(n) = 2(2t(\frac{n}{4}) + c_2 \frac{n}{2})$$

Observing the pattern developing yields:

$$T(n) = 2^i T(\frac{n}{2^i}) + ic_2 n$$
$$T(n) = 2^k T(\frac{n}{2^k}) + kc_2 n$$
$$= 2^k T(1) + c_2 n \log n$$

This is based on knowing that $2^k = n$ and that $\log 2^n = k$. Note that since this is CS and not math, the '2' in $\log_2$ is implicit and therefore not written.

8.1.4 Quicksort

Quicksort is another recursive sorting algorithm. Start by picking a pivot point, then move all the items lower than the pivot to the left, and all items higher than it to the right. Then execute quicksort on the left and right lists. Here, have some code:

```
def quicksort(A, p, r):
    if p < r:
        q = partition(A, p, r)
        quicksort(A, p, q-1)
        quicksort(A, q+1, r)
```

Note: this code is a pretty poor representation of how quicksort really works.

8.1.5 Heapsort

Heapsort is an in-place comparison-based sort invented by J.W.J. Williams in 1964. It's usually slower than quicksort, but it always runs in $O(n \log n)$ time. Heapsort gets its name from the underlying data structure, a *heap*.

Heaps are a special variant of binary trees in that they are constructed such that they satisfy the heap property. They are supposed to be balanced[2] and left-justified, in order to be most efficient. The heap property itself just states that a parent node is no smaller than its child nodes. Another version (this one is for *max heaps*) is that the parent value may be no greater than its children.

This is all nice if you're using a linked-list style tree, but for an in-place heapsort, you'll need to map a heap into an array... somehow. One way of doing it is to say that the left child of index i is at index $2i + 1$, the right child of index i is at index $2i + 2$, and the parent is at $\lfloor \frac{i-1}{2} \rfloor$.

Heapsort, then, works by first organizing all the data points into a heap, then withdrawing the root element and placing it at the larger end of the sorted list. Since there is now no more root note, swap the deepest, rightmost node into its place. Since the heap property is now probably broken, go back through and re-heapify everything. Since both the initial sort and the re-heapify operation runs in $\log n$ time, and re-heapify is run n times, the algorithm obviously runs in $n \log n$ time.

[2] Trees are considered balanced if all the nodes at depths 0 to $n - 2$ have two children.

8.2 Searches

8.2.1 Binary Search

A binary search relies on usage of binary search trees. These trees are regular binary trees, but they are required to satisfy a few conditions:

- An element's right child must be larger than the element.
- An element's left child must be smaller than the element.

Using a constructed binary search tree, it is possible to find a value by simple traversal of the tree. Given a root node and a key, check if the key is equal to the key at the node; if so, return it. Otherwise, check if the key is less than or greater than the node's key; if less than, recurse into the left child, otherwise, recurse into the right child. If no appropriate child can be found, return nothing. A similar process is used for insertion, too.

8.3 Pathfinding Algorithms

8.3.1 Warshall's Algorithm

Warshall's algorithm works on a graph using an adjacency matrix. Using this information it is possible to calculate what nodes are reachable from another given node, the information for which can be recorded into another kind of matrix, where a '1' indicates that a path can be found from one node to the other – this is called a transitive closure matrix. Warshall's algorithm in an algorithm that calculates these transitive closure matrices using adjacency matrices.

The algorithm works by slowly adding edges between nodes as they are calculated, iteratively. The first iteration is just the adjacency matrix, with 1s representing the existence of paths with no intermediate vertices. Boxed rows and columns are used to get future iterations. On the second iteration, 1s represent the existence of paths with intermediate vertices numbered not higher than 1. The third iteration keeps the same principle, with intermediate vertices numbered not higher than 2.

```
def Warshall(A):
    R[0] = A
    for k in range(len(A)):
        for i in range(len(A)):
            for j in range(len(A)):
                R[k][i][j] = INCOMPLETE
```

Didn't have enough time to copy down the algorithm. Anyways, this thing runs in $O(n^3)$ time and space. Improvements to space efficiency can be made if you don't save as many old versions R of the adjacency matrix. It's generally not a good idea to use Warshall's in an undirected graph, BFS and DFS will do the same job much quicker.

8.3.2 Floyd's Algorithm

Floyd's algorithm is essentially the same as Warshall's algorithm, to such an extent that the algorithm is called the Floyd-Warshall algorithm. There's no particular reason to cover it here.

8.3.3 Dijkstra's Algorithm

Dijkstra's algorithm is a simple algorithm for finding the shortest path between two points on a graph, using dynamic programming and a method similar to breadth-first-search. It works by first exploring the group of nodes around the starting node and assigning them a distance value, based on how far the algorithm had to go (accounting for edge weights) in order to get there. Each node is parented to the node that explored it, so the first round of nodes are all parented to the base node. From there the algorithm selects the next nearest node, explores its neighbors, and adds them onto the list of nodes that could be explored. The process is

repeated until one of the explored nodes is found to be the destination, at which point following back the chain of parented nodes will reveal the shortest path to the destination.

Dijkstra's algorithm generally requires non-negative edge weights. If all edge weights are the same, it's probably a better idea to use use breadth-first-search instead. If you remove the stop condition and instead just tell the algorithm to stop searching once it's ran of of nodes to search through, it can instead be used to generate a list of shortest paths from the start node to *all* nodes.

The complexity analysis of this algorithm is fairly simple. The initialization step runs in O(V) time, where V is the number of vertices. Inserting into the queue of nodes to be explored can be run in O(V log V) (or maybe lower if you use a minimum heap for a priority queue).

8.3.4 DAG Algorithm

Directed acyclic graphs are directed graphs that contain no cycles. They have the special property of being able to be topologically sorted, i.e. sorted in a linear order such that all nodes are forward nodes. If you choose a source node in a DAG, any nodes left of the source wil have infinite cost, since they cannot be reached. Finding the shortest path in a topologically sorted graph is actually very simple: For each vertex u taken in topologically sorted order, and for each v in vertexes adjacent to u, relax(u, v).

8.4 Misc

8.4.1 Tower of Hanoi

This is an example of recursive algorithms. The legend goes that a group of Eastern monks are the keepers of three towers on which 64 golden rings sit. The monks are to move the rings from the first tower to the third tower one at a time, but never moving a larger ring on top of a smaller ring (the rings are ordered from top to bottom in ascending order by size). Once all 64 rings have been moved, the world will come to an end. Sadly, this has nothing to do with quantum bogosort, and it probably doesn't go back that far either — it was probably invented by a French mathematician in the 19^{th} century.

This problem can be solved using a recursive algorithm, looking something like the following:

```
def MoveTower(disk, source, dest, spare):
    if disk == 0
        move disk from source to destination
    else:
        MoveTower --- NOPE
```

And here's the process of solving it using the recursive formula:

$$
\begin{aligned}
m(n) &= 2m(n-1)+1, m(1)=1 \\
m(n-1) &= 2m(n-2)+1 \\
m(n) &= 2(2m(n-2)+1)+1 \\
&= 4m(n-2)+2+1 \\
m(n-2) &= 2m(n-3)+1 \\
&= 4(2m(n-3)+1)+2+1 \\
&= 8m(n-3)+4+2+1 \\
&= 2^i(n-i)+2^i-1 \\
m(1) &= 1 \\
n &= 1 \\
i &= n-1 \\
i &= 0 \\
&= 2^{n-1}m(n-(n-1))+2^{n-1}-1 \\
&= 2^{n-1}m(1)+2^{n-1}-1 \\
&= 2^{n-1} \\
&= O(2^n)
\end{aligned}
$$

The tower of Hanoi algorithm, then, runs in exponential time.

8.4.2 Huffman Coding Algorithm

The algorithm for Huffman coding uses binary trees and weights to construct a Huffman tree. It's actually very simple. Given a list of symbols and their weights, combine the two lowest weights into a tree with their summed weight at the top and the two values at the leaves. Repeat until all values are contained within one binary tree. Once that's done, assign a "1" to each right branch and a "0" to each left branch. By following a path down the tree, you can find the coding for a particular value or symbol.