Exercise/Problem Guide for *Nature of Computation* by Moore and Mertens

Muthu Chidambaram

Last Updated: August 12, 2019

Contents

About

"Computer science is no more about computers than astronomy is about telescopes."

- Edsger Dijkstra (Maybe)

These notes contain my solutions to some exercises and problems from the book *Nature of Computation* by Christopher Moore and Stephan Mertens.

1 Prologue

1.1 Exercises

1.1.1 Exercise 1

The first graph has exactly 2 vertices of odd degree, so it has a Eulerian path. The second graph has 4 vertices of odd degree, so it does not.

1.2 Problems

1.2.1 Problem 1

Each edge in a finite graph increases the total sum of the graph's vertex degrees by 2. Thus, this sum must be an even number, so we cannot have an odd number of vertices with odd degree (otherwise the sum would be odd).

1.2.2 Problem 2

Consider a finite simple graph with n vertices. Each vertex can have a degree between 1 and $n-1$, since we cannot have multiple edges or self-loops. Thus, by the pigeonhole principle, at least two of the n vertices must have the same degree (there is no bijection between the n vertices and the $n-1$ degree options).

1.2.3 Problem 3

If every vertex has even degree, we can construct a set of covering cycles as follows. Choose an arbitrary vertex with non-zero degree and traverse a cycle starting at that vertex, while deleting all edges traversed along the way. Repeat this procedure until all edges have been deleted. We are guaranteed to be able to find the aforementioned cycles since the graph is connected; an edge ”leaving” a vertex can be paired with an edge ”coming in” to the vertex.

Once we have a set of covering cycles, we can combine them into a single cycle as follows. Consider a cycle starting at vertex a and another, edge-disjoint cycle starting at vertex b such that b also occurs in the cycle starting at a. We can then combine the two cycles by starting at vertex a and traversing its cycle until arriving at vertex b, at which point we traverse the cycle starting at vertex b. Finally, we finish the rest of vertex a's cycle from b onwards. Since cycles a and b were edge-disjoint, this combined cycle does not visit any edge twice. We can repeat this cycle combination procedure until only a single cycle is left, which is the Eulerian cycle.

2 The Basics

2.1 Exercises

2.1.1 Exercise 2.1

Let $a = nb + k$. If $n \geq 2$, then $\frac{a}{2} \geq b > a \mod b$ since $a \mod b \leq b - 1$. For $n = 1$ we have

$$k < b \implies \frac{k}{2} < \frac{b}{2} \implies k < \frac{b}{2} + \frac{k}{2} = \frac{a}{2}$$

Since Euclid's algorithm "alternates" between performing divisions on a and b (since a is replaced by b after a step), the number of divisions it performs is bounded by $2\log_2 a$.

2.1.2 Exercise 2.2

The maximum divisor of a is bounded by $\sqrt{a}$. We can find all primes less than or equal to $\sqrt{a}$ in polynomial time by using a prime number sieve approach (e.g. Sieve of Eratosthenes). Once we have the primes, the number of operations it takes to compute the prime factorization of a is bounded by $\sqrt{a}\log_2 a$, since the number of primes is bounded by $\sqrt{a}$ and 2 is the smallest prime.

2.1.3 Exercise 2.3

Changing logarithm base amounts to multiplying by a constant.

2.1.4 Exercise 2.4

We have the following

- $4\log_2 n = \log_2 n^4 \implies n^4$
- $4\sqrt{n} = \sqrt{16n} \implies 16n$
- $4n \implies 4n$
- $4n^2 = (2n)^2 \implies 2n$
- $4 * 2^n = 2^{n+2} \implies n + 2$
- $4 * 4^n = 4^{n+1} \implies n + 1$

2.2 Problems

2.2.1 Problem 2.18

An adjacency matrix requires a single bit for each vertex pair, so we need n^2 bits to specify it. A list of edges consists of m tuples of the form (a, b) for $1 \leq a, b \leq n$, so it requires $2m\log n$ bits. Thus, we are better off using the adjacency list format for sparse graphs and the adjacency matrix format for dense graphs. If, however, we consider multigraphs, then the adjacency matrix requires $n^2 \log n$ bits (as we need to store the number of edges from a to b). Thus, in the multigraph case, we can always choose to use adjacency lists.

2.2.2 Problem 2.19

Suppose $f(n) = O(n^a)$ and $g(n) = O(n^b)$. Then we have that

$$\begin{aligned} &\exists c_1, c_2, N \mid f(n) \leq c_1 n^a,\ g(n) \leq c_2 n^b \quad \forall n \geq N \\ &\implies g(f(n)) \leq g(c_1 n^a) \leq c_2 (c_1 n^a)^b = O(n^{ab}) \end{aligned}$$

so poly(n) is closed under composition. Thus, any polynomial time algorithm that calls another polynomial time algorithm remains polynomial.

2.2.3 Problem 2.20

Since $f(n) = 2^{\Theta(\log^k n)}$ and $g(n) = O(n^a)$, there exists N such that for all $n \geq N$ we have $f(n) \geq 2^{c_1 \log^k n}$ and $g(n) \leq c_2 n^a$ for some constants c_1, c_2. Thus,

$$\begin{aligned} \lim_{n\to\infty} \frac{f(n)}{g(n)} &\geq \lim_{n\to\infty} \frac{2^{c_1 \log^k n}}{c_2 n^a} \\ &= \lim_{n\to\infty} \frac{(n^{c_1})^{\log^{k-1} n}}{c_2 n^a} \end{aligned}$$

Since $\lim_{n\to\infty} \log^{k-1} n = \infty$, we have that $f(n) = \omega(g(n))$. To see that $f(n) = o(h(n))$, we substitute $n = 2^x$ to get

$$\begin{aligned} \lim_{x\to\infty} \frac{f(2^x)}{g(2^x)} &\leq \lim_{x\to\infty} \frac{2^{a_1 x^k}}{2^{a_2 2^{cx}}} \\ &= 0 \end{aligned}$$

Where we used the fact that $\lim_{x\to\infty} \frac{a_1 x^k}{a_2 2^{cx}} = 0$. Finally, to see that QuasiP is closed under composition, we can check that

$$2^{\log^{k_1} 2^{\log^{k_2} n}} = 2^{\log^{k_1 k_2} n}$$

3 Insights and Algorithms

3.1 Exercises

3.1.1 Exercise 3.1

Assume $f(n) = 2^n - 1$. Then $f(n+1) = 2^{n+1} - 2 + 1 = 2^{n+1} - 1$.

3.1.2 Exercise 3.2

We have that

$$\begin{aligned}(QQ^*)_{ab} &= \frac{1}{n}\sum_{k=0}^{n-1} w_n^{ak}\overline{w_n^{kb}} \\ &= \frac{1}{n}\sum_{k=0}^{n-1} w_n^{k(i-j)}\end{aligned}$$

If $i = j$, then $(QQ^*)_{ab} = 1$. Otherwise, since $w_n^{(i-j)}$ is a root of unity,

$$(QQ^*)_{ab} = w_n^{i-j}(QQ^*)_{ab} \implies (1 - w_n^{i-j})(QQ^*)_{ab} = 0 \implies (QQ^*)_{ab} = 0$$

And we have that $Q^* = Q^{-1}$ as desired.

3.1.3 Exercise 3.3

Exercise seems ambiguous - if we're just talking about $f(1)$, then we can simply add an if case for it. Otherwise, we can return both f_min and the index j. I don't think it's possible to reconstruct full scores from just returning the index by itself (without wasting computation).

3.1.4 Exercise 3.4

If we let $g(n)$ be the number of calculations for $f(n)$ after calling $f(1)$, then we have by definition of the algorithm that $g(n) = \sum_{k=1}^{n-1} g(k)$ (since every call to $f(k)$ calls $f(n)$). We see that $g(2) = g(1) = 2^0$, and we assume that $g(n) = 2^{n-2}$. Then we have that

$$g(n+1) = \sum_{k=1}^{n} g(k) = g(n) + \sum_{k=1}^{n-1} g(k) = 2g(n) = 2^{n-1}$$

So we are done by induction.

3.1.5 Exercise 3.5

Suppose we pick a subset of j characters from both s and t. Then there is a unique alignment that corresponds to assigning the subsets to one another (in the order of characters) and then deleting/inserting everything else that's not aligned. This also covers all possible alignments, so we have that the number of alignments is $\binom{2n}{n}$.

3.1.6 Exercise 3.6

There are n choices for where to cut s to make s' and n choices for where to cut t to make t', hence $O(n^2)$.

3.1.7 Exercise 3.7

Calculating optimal edit distance finds an optimal alignment in the process; we only need to also return with $d(s,t)$ the choice of operation that was used and then compose these operations across subproblems.

The edit distance problem is available to solve on LeetCode. My solution is as follows

```
def minDistance(self, word1, word2):
    N, M = len(word1), len(word2)
    DP = [[0] * (M+1) for i in range(N+1)]
    for j in range(M):
        DP[N][j] = M - j
    for i in range(N):
        DP[i][M] = N - i
    for i in range(N-1, -1, -1):
        for j in range(M-1, -1, -1):
            not_eq = 1
            if word1[i] == word2[j]:
                not_eq = 0
            DP[i][j] = min(DP[i][j+1] + 1,
                           DP[i+1][j] + 1,
                           DP[i+1][j+1] + not_eq)

    return DP[0][0]
```

3.1.8 Exercise 3.8

If there is a path $s \to t$, then there must be a product of the form $A_s i A_i j ... A_k t$ that is non-zero and contains a maximum of $n-1$ terms. Any such product can be extended to $n-1$ terms if we introduce self-loops (since $A_i i = 1$), so we have that $(1+A)^{n-1}_{st}$ is non-zero. The reverse direction can be shown similarly.

3.1.9 Exercise 3.9

As hinted, we can maintain an array $V[i][j]$ that tracks which middle vertex k was used to get the minimum $B_{ij}(\log_2 n)$. Then, we can reconstruct the optimal path backwards by looking at $V[i][j] = k_1, V[i][k_1] = k_2, ...$ until we get to i.

3.1.10 Exercise 3.10

If there is a cycle whose total length is negative, then there is no fixed point (since we can repeatedly go through that cycle to decrease distance). Otherwise, there is no issue, since there is no way to reduce distance by visiting a vertex more than once.

3.1.11 Exercise 3.11

By definition, $B_{ij}(m)$ must be an upper bound on the length of the shortest path from i to j, as otherwise we would be positing that there exists k such that the shortest path from i to k to j is shorter than the shortest path from i to j. The term $B_{ij}(m)$ is the composition of paths $Bik(m-1)$ and $B_{kj}(m-1)$, so the number of steps in $B_{ij}(m)$ is at most twice the maximum of the number of steps in $B_{ik}(m-1)$ (over all k). From this it is clear that the number of steps after m outermost iterations is at most 2^m (since $m = 0$ corresponds to a single step). If there are no negative cycles, then the shortest path between two vertices will take at most $n - 1$ steps, so we only need to iterate up to $m = \log_2 m$.

3.1.12 Exercise 3.12

Suppose we complete the for loop and there are two vertices that are not connected. Since the for loop considers every edge, there must be no path between these two vertices. However, this contradicts the assumption of the graph being connected, so the final result of the for loop must be a spanning tree.

3.1.13 Exercise 3.13

For $n = 2$, it is clear that a forest with $n - 1$ edges must be a spanning tree. Now we assume the same is true for n and consider a forest with $n+1$ vertices. By the inductive assumption, any sub-forest consisting of $n - 1$ edges must be a spanning tree for the n vertices it connects. Since these n vertices are already connected, adding an edge between any of them would introduce a cycle. Thus, the only way to grow this forest of n vertices to a forest of $n + 1$ vertices is to add an edge to vertex $n+1$, so a forest with n edges must be a spanning tree for its $n + 1$ vertices. This proves one direction; the other direction can be proved similarly.

3.1.14 Exercise 3.14

Just add the bolded edges in Figure 3.16 in order of weight.

3.1.15 Exercise 3.15

The proof idea is the same as that of Lemma 3.1, except now replacing an edge e with another edge e' cannot preserve a minimum spanning tree, since $e \neq e'$.

3.1.16 Exercise 3.16

Run Kruskal's but negate the edges in the graph. This has to be the maximum, since we already proved Kruskal's to be optimal.

3.1.17 Exercise 3.17

The first axiom is vacuously true for linear independence. For the second axiom, if Y were not linearly independent then we could extend the nontrivial linear relation sending Y to 0 to X, so the second axiom must also be true. The third axiom holds similarly.

3.1.18 Exercise 3.18

Suppose we obtain a maximum flow using Ford-Fulkerson. At each stage of Ford-Fulkerson, we increase the current flow f by the minimum $c_f(e)$ in an augmenting path. Since the capacities are all integers, this means at each stage we increase f by an integral amount, so there is a max flow obtained via Ford-Fulkerson that is an integer.

3.1.19 Exercise 3.19

No, the maximal flow need not be unique; consider a graph with only a single edge from the source to a fork consisting of edges with equal capacity and connecting back to a single node (looks like a kite). The difference between two max flows must be 0, since there must be some edge that they both share (otherwise we could just combine both two get a larger flow).

3.1.20 Exercise 3.20

The min cut is not unique; we can remove the two edges from the source or we could remove the two edges to the target. Additionally, we can also remove the lower edge from the source and the upper edge to the target.

3.1.21 Exercise 3.21

If there is a flow of value m, it must necessarily pass through m of the original edges in the bipartite graph G, since they were all assigned capacity 1. A matching consisting of m edges in G corresponds to m unique left nodes (connected to s) and m unique right nodes (connected to t), so we can send a flow of m through this matching.

3.2 Problems

3.2.1 Problem 3.1

We see that $n = 1$ requires only the single move of moving the one disk directly to the desired peg, and that $n = 2$ cannot be done in fewer than the 3 moves indicated by the recursive algorithm. Suppose the recursive algorithm is optimal for n. To move $n+1$ disks, we must move disk $n+1$ to the bottom of the desired peg. Thus, we first move n disks to the non-desired peg, which requires $2^n - 1$ moves. Now there is no "better" move than moving disk $n + 1$ to the desired peg, since it will be put in its final spot. After making this move, we again have to move the n disks from the other peg to the desired peg, which takes another $2^n - 1$ moves, for a total of $2(2^n - 1) + 1 = 2^{n+1} - 1$ moves, as desired.

3.2.2 Problem 3.2

Figure 3.24 is very helpful. The recursive solution for Towers of Hanoi can be translated to finding a Hamiltonian path on an n-dimensionl cube as follows

1. Find a Hamiltonian path on one face of the cube, which is itself an $n-1$-dimensional hypercube.
2. Now move from this face to the opposite face; this requires moving once along a single edge connecting the two faces (this is how we create a hypercube in the first place).
3. Now we can find a Hamiltonian path on the opposite face, and we are done.

The vertices of the cube can be identified with different Hanoi states, and the edges can be identified with moves.

4 Needles in a Haystack: the Class NP

4.1 Exercises

4.1.1 Exercise 1

If a 2-coloring of a graph exists, then we can flip all of the colors in the graph to produce another valid two-coloring. Thus, we can proceed to construct a two-coloring by picking an uncolored vertex and then coloring it an arbitrary color, which then determines the colors of all of the vertices it is connected to. We do this until either the graph is completely colored, or until we need to recolor an already colored vertex (in which case the graph is not 2-colorable). This algorithm is linear in the number of edges and vertices of the graph, so 2-coloring is in P.

4.1.2 Exercise 2

4.1.3 Exercise 3

Suppose we wish to determine whether a graph G with n vertices is k-colorable. We can extend G to be a graph G' with $n+1$ vertices by adding an additional vertex that we then connect to all of the original n vertices of G (this can be done in $O(n)$ time). By construction, G' is $(k+1)$-colorable if and only if its subgraph G is k-colorable (since node $n+1$ is connected to all of G, the subgraph G must be colored with only k colors).

4.1.4 Exercise 4

Since boolean operations are commutative, we can write ϕ' as:

$$\begin{aligned}\phi'(p,q,r) &= (p \vee q) \wedge (p \vee \bar{q}) \wedge (r \vee q) \wedge (r \vee \bar{q}) \wedge (\bar{p} \vee \bar{r}) \\ &= p \wedge r \wedge (\bar{p} \vee \bar{r})\end{aligned}$$

which is clearly not satisfiable, since $p = 1, r = 1 \implies (\bar{p} \vee \bar{r}) = 0$.

4.1.5 Exercise 5

For a graph to be 3-colorable (let our 3 colors be R, G, and B), each vertex in the graph must be colored either R, G, or B, and no adjacent vertices may have the same color. If we let $v_r^{(i)}, v_g^{(i)}, v_b^{(i)}$ denote boolean variables that are true when vertex i is colored R, G, and B respectively and $N(i)$ denote all of the neighbors of vertex i, we can translate 3-colorability into the following boolean expression for each vertex i:

$$\bigwedge_i (v_r^{(i)} \vee v_g^{(i)} \vee v_b^{(i)}) \wedge \left((v_r^{(i)} \wedge_{j \in N(i)} \bar{v}_r^{(j)}) \vee (v_b^{(i)} \wedge_{j \in N(i)} \bar{v}_b^{(j)}) \vee (v_g^{(i)} \wedge_{j \in N(i)} \bar{v}_g^{(j)})\right)$$

Repeatedly applying the distributive law to the part of the above expression that consists of ors of ands produces a CNF expression that must be true for every vertex i. Since satisfiability of the resulting expression corresponds directly to whether the original graph is 3-colorable or not, we have that 3-colorability can be reduced to SAT.

4.1.6 Exercise 6

Factoring out x in the first expression leads to $x \vee (z_1 \wedge \bar{z}_1) = x$. The second expression is even more straightforward; we just factor out $(x \vee y)$.

4.1.7 Exercise 7

If any of the x_i are true, then it is clear that the right-hand side can be made true by picking (z_1, z_2) such that the (at most) two clauses not containing a true x_i value can be made true. For the other direction, iterating through all possible (z_1, z_2) values shows that the right-hand side is only true if at least one of the x_i (where i depends on the values of z_1, z_2) is true.

There is probably a better approach here than the above "brute force" reasoning; I should come back to this.

4.1.8 Exercise 8

The approach from Exercise 5 still works.

4.1.9 Exercise 9

There are only $(2n)^k$ possible unique k-clauses (each of the k literals can either be one of the variables or its complement), so the number of bits needed to specify a satisfiability problem is polynomial in the number of variables (each of which requires only a single bit).

5 Appendix

5.1 Exercises

5.1.1 Exercise A.1

We can choose $n_3 = max(n_1, n_2)$ such that $f_1(n)+f_2(n) \leq (C_1+C_2)g$ whenever $n > n_3$.

5.1.2 Exercise A.2

Similar to the first exercise, but now we have C_1C_2h instead.

5.1.3 Exercise A.3

Logarithms of different bases differ by a constant.

5.1.4 Exercise A.4

The argument treats k as a constant, when in reality $k = O(n)$. The answer should be $O(n^3)$, which can be checked by looking at the closed form of the sum.

5.1.5 Exercise A.5

$2^{O(n)}$ and $O(2^n)$ are not the same, since $\limsup_{n\to\infty} \frac{2^{C_1 n}}{C_2 2^n}$ only converges to a nonzero constant when $C_1 = 1$.

5.1.6 Exercise A.6

$n^k \neq \Theta(2^n)$, $e^{-n} \neq \Theta(n^{-c})$, and $n! \neq \Theta(n^n)$, as all limits go to 0.

5.1.7 Exercise A.7

Take $f = n$ and $g = 2n$.

5.1.8 Exercise A.8

1. We can pull out the exponents as constants, so $f = \Theta(g)$.
2. $3 < 2^2$ so $f = o(g)$.
3. $n = \omega(\log^2 n)$ so $f = \omega(g)$.
4. $2^{\log n} \to \infty$ so $f = \omega(g)$.

5.1.9 Exercise A.9

One example is $n^{\log n}$.