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

Muthu Chidambaram

Last Updated: June 23, 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 Insights and Algorithms

1.1 Exercises

1.1.1 Exercise 3.1

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

1.1.2 Exercise 3.2

We have that

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

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.

1.1.3 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 $f_m in$ 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).

1.1.4 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.

1.1.5 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 $\binom{2n}{n}$.

1.1.6 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 $O(n^2)$.

1.1.7 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]
```

1.1.8 Exercise 3.8

If there is a path $s \to t$, then there must be a product of the form $A_siA_ij...A_kt$ 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 $A_ii = 1$), so we have that $(1+A)_{st}^{n-1}$ is non-zero. The reverse direction can be shown similarly.

1.1.9 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.

1.1.10 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.

1.1.11 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 $m = \log_2 m$.

2 Appendix

2.1 Exercises

2.1.1 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 $n > n_3$.

2.1.2 Exercise A.2

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

2.1.3 Exercise A.3

Logarithms of different bases differ by a constant.

2.1.4 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.

2.1.5 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 $C_1 = 1$.

2.1.6 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.

2.1.7 Exercise A.7

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

2.1.8 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 $f = \omega(g)$.

2.1.9 Exercise A.9

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