Solutions to Homework 2 Practice Problem

[DPV] Problem 2.1 – Practice Multiplication

The product is 28,830. See ***2.1 Practice Multiplication.pdf*** for the value calculated at each level of the recursive stack.

[DPV] Problem 2.5 – Recurrence

Solutions:

(a) $T(n) = 2T(n/3) + 1 = O(n^{\log_3 2})$ by the Master theorem

(b) $T(n) = 5T(n/4) + n = O(n^{\log_4 5})$ by the Master theorem

(c) $T(n) = 7T(n/7) + n = O(n \log_7 n) = O(n \log n)$ by the Master theorem

(d) $T(n) = 9T(n/3) + n^2 = O(n^2 \log_3 n) = O(n^2 \log n)$ by the Master theorem

(e) $T(n) = 8T(n/2) + n^3 = O(n^3 \log_2 n) = O(n^3 \log n)$ by the Master theorem

(f) $T(n) = 49T(n/25) + n^{3/2} \log n = O(n^{3/2} \log n)$

Hint: the contribution of level i of the recursion is $(\frac{49}{125})^i O(n^{3/2} \log n)$.

(g) $T(n) = T(n-1) + 2 = O(n)$

This can be found through substitution:

$$\begin{aligned} T(n) &= T(n-1) + 2 \\ T(n) &= (T((n-1)-1) + 2) + 2 \\ T(n) &= T(n-2) + 2 + 2 \end{aligned}$$

This pattern repeats n times until $T(n-n) = T(0)$; on the right we have $2n = O(n)$

(h) $T(n) = T(n-1) + n^c = \sum_{i=1}^{n} i^c + T(0) = O(n^{c+1})$

$$\begin{aligned} T(n) &= T(n-1) + n^c \\ T(n) &= (T((n-1)-1) + (n-1)^c) + n^c \\ & \quad T(n-2) + (n-1)^c + n^c \qquad \textit{we simplify} \\ T(n) &= (T((n-2)-1) + (n-2)^c) + (n-1)^c + n^c \\ & \quad (T((n-3) + (n-2)^c) + (n-1)^c + n^c \end{aligned}$$

This pattern repeats n times until $T(n-n) = T(0)$; the $(n-1)^c$ resolves to a polynomial where n^c is the dominating term. This repeats n times, giving $O(n * n^c) = O(n^{(c+1)})$

(i)

$$T(n) = T(n-1) + c^n = \sum_{i=1}^{n} c^i + T(0) = O(c^n)$$

This can be solved similar to (h); This pattern repeats n times until $T(n-n) = T(0)$; we have a series $c^0 + c^1 + \ldots + c^n$, the last term dominates giving $O(c^n)$

(j)

$$T(n) = 2T(n-1) + 1 = \sum_{i=0}^{n-1} 2^i + 2^n T(0) = O(2^n)$$

(k)

$$T(n) = T(\sqrt{n}) + 1 = \sum_{i=0}^{k} 1 + T(b)$$

where k is an integer such that $n^{\frac{1}{2^k}}$ is a small constant b (the size of the base case). This implies that $k = O(\log \log n)$ and $T(n) = O(\log \log n)$.

[DPV] 4.11 (length of shortest cycle on a graph)

As presented, we are given a ***directed graph*** with ***positive edge weights***. These conditions assure that there is no negative-weight cycle within the graph. How do we find the length of the shortest cycle? Note that the shortest cycle visiting vertices u and v has length $d(u, v) + d(v, u)$, hence the answer we are looking for is

$$\min_{u,v}(d(u, v) + d(v, u))$$

Notice that if the quantity above is equal to infinity it means the graph is acyclic, and we report that.

Solution:

(a) Algorithm:
Run `Floyd-Warshall` on the given graph to get an array $\{d(u, v)\}_{u,v \in V}$ of the distances between any pair of points. It is assumed that at the beginning any (u, v) pair which is not represented by an edge $[(u, v) \notin E]$ is set $d(u, v) = \infty$. Now, inspect the diagonal in the array and our shortest path is the $\min(d(v, v))$ across all vertex $v \in V$

(b) Justification of Correctness:
To see why this algorithm is correct: note first that every cycle visiting vertices u and v can be broke into two directed paths: from u to v and from v back to u. Each such path achieves its minimum length at $d(u, v)$ and $d(v, u)$ respectively, hence the cycle of minimal length is given by this sum.

(c) Runtime Analysis:
`Floyd-Warshall` takes $O(|V|^3)$ and the minimum can be found in $O(|V|)$, leading to an $O(|V|^3)$ runtime.

[DPV] 4.21 (Currency trading)

We are presented with a directed graph problem, so hopefully we can use a known algorithm as the basis for our solution. The key is to recognize two things: (a) the currency calculation is a product (b) we want to maximize that product. We solve for the former by converting the exchange rates to logs (recalling that $\log(a) + \log(b) = \log(a \times b)$), and solve for the latter by using the negative of the log for our edge weight (flipping maximization to minimization).

Solution:

(a) Algorithm:
For part (a), we create a graph where the vertices represent countries, and the edges between them have a weight $w_{i,j} = -\log r_{i,j}$. We then run Bellman-Ford from vertex s to vertex t, and the minimal weight path represents the most advantageous sequence of currency trades.

For part (b), we run one more iteration of Bellman-Ford, and if any of the distances (weights) change, we have detected a negative cycle – such a cycle represents the trading anomaly which allows for infinite profit.

(b) Justification of Correctness:
Why does this work? By converting the exchange rates to logs and then negating that value the weight of the shortest path (the sum of the negated log values) will represent the series of exchanges which maximizes the product of the exchange rates. Note that since edge weights are negative we cannot use Dijkstra's.

(c) Runtime Analysis:
In part (a) we create our graph of $|V|$ vertices representing the n currencies, and $|E|$ edges representing the various exchange rates. We then run Bellman-Ford which takes $O(|V||E|)$ time. Part (b) is simply one additional iteration of Bellman-Ford, $O(|E|)$ updates to process each edge again.

[DPV] Binary Search Modified

Design an O(log(n)) algorithm to find the smallest missing natural number in a given sorted array. The given array only has distinct natural numbers. For example, the smallest missing natural number from $A = \{3, 4, 5\}$ *is 1 and from* $A = \{1, 3, 4, 6\}$ *is 2.*

Solution:

(a) Algorithm:

If we have an array of length n we expect the natural numbers $\{1, 2, 3, \ldots, n\}$ to be in the array. Since at least one number is missing this means there is at least one number such that its position (index) does not match its value.

To solve our problem we will modify Binary Search in the following way. Once the midpoint index is found, we check if the value at the midpoint is equal to the midpoint; that is, does $A[mid] = mid$. If it does, that tells us all the expected values to the left of the midpoint are in place, so we recurse on the right half of the array. If the value found is greater then the midpoint, we know there is at least one missing value to the left, so we recurse on that half.

Our base case is when we get to a single value. If that value, $A[i]$ equals it's index i, the first missing value is $i + 1$. If the value does not equal the index, the missing value is i. We return the missing value.

(b) Justification of Correctness:

Why does this work? Our input is natural numbers sorted in non-decreasing order, which allows us to take advantage of Binary Search. As natural numbers begin at 1, then the missing value is the lowest i such that $A[i] \neq i$. When we check if $A[mid] = mid$, if they match we know all the values $1 \ldots mid$ are present. If they don't match, we know some value in the range $1 \ldots mid - 1$ is missing.

(c) Runtime Analysis:

Our modification to Binary Search simply changes the comparison and return value. Both changes remain constant time operations, so there is no impact to the original $O(\log n)$ runtime. We can convince ourselves of this by considering the following: at each level of recursion we perform one constant time comparison and then solve a single recursive problem on just one-half of the current input. This gives the recurrence relation $T(n) = T(\frac{n}{2}) + O(1)$ which solves to $O(\log n)$ by the Master Theorem (where $a = 1$, $b = 2$, and $d = 0$).