

CSE 101: Homework 3

(Due Thursday, May 8th in Lecture)

4.21(a) We solve the problem by creating a graph $G = (V, E)$ with $V = \{c_1, \dots, c_n\}$, and edges $e_{ij} = -\log r_{ij}$. We then run the Bellman-Ford algorithm on this graph with input s , and return the shortest path from s to t . We must show that the shortest path returned is in fact the most advantageous sequence of currency exchanges to get from s to t . Let P_{st} be the set of paths from s to t in G . Our algorithm returns

$$\begin{aligned} \arg \min_{p \in P_{st}} \sum_{(i,j) \in p} -\log r_{ij} &= \arg \max_{p \in P_{st}} \sum_{(i,j) \in p} \log r_{ij} \\ &= \arg \max_{p \in P_{st}} \prod_{(i,j) \in p} r_{ij} \end{aligned}$$

So the shortest path is equivalent to the path with best exchange rate. It is worth knowing that G cannot have negative cycles, otherwise there would be an opportunity to make infinite amount of money.

4.21(b) Assume there exists some cycle C that has $\prod_{(u,v) \in C} r_{uv} > 1$. This is equivalent to saying $-\sum_{(u,v) \in C} \log r_{uv} < 0$, or in other words that a negative cycle exists in the G from part (a). So, we must devise an algorithm that finds a negative cycle in G if it exists. We know an algorithm for doing such a task (see page 119 last paragraph of 4.6.2), and the proof of correctness is that of the proof of correctness for Bellman-Ford.

2.5(a) Here $a = 2, b = 3, d = 0$, so $T(n) = O(n^{\log_3 2})$ by the Master's Theorem on page 49.

2.5(g) Using iteration to solve $T(n) = T(n-1) + 2 = T(n-2) + 4 = \dots = T(0) + 2n = O(n)$.

2.5(h) Again, using iteration gives $T(n) = T(n-1) + n^c = T(n-2) + 2n^c = \dots = T(0) + n^{c+1} = O(n^{c+1})$.

2.5(k) Let $n = 2^m$. Then $T(2^m) = T(2^{m/2}) + 1$. Let $R(m) = T(2^m)$. Then, $R(m) = R(m/2) + 1$. We know from the master's theorem that $R(m) = O(\log m)$. Substituting back in for n gives $T(n) = R(m) = O(\log m) = O(\log \log n)$.

2.12 Let $f(n)$ have running time $T(n)$. The recurrence for the runtime of this function is $T(n) = 2T(n/2) + O(1)$. Using the Master's theorem we have $a = 2, b = 2, d = 0$ which gives $T(n) = O(n^{\log_2 2}) = O(n)$.

2.17 The idea here is to examine the element at $A[\lceil n/2 \rceil]$ recursively.

DQElementEqIndex($A[1 \dots n], offset$):

1. if $A[\lceil n/2 \rceil]$ equals $offset + \lceil n/2 \rceil$, then return TRUE
2. if $|A| \leq 1$, return FALSE
3. if $A[\lceil n/2 \rceil] < offset + \lceil n/2 \rceil$, return **DQElementEqIndex**($A[(\lceil n/2 \rceil + 1) \dots n], offset + \lceil n/2 \rceil$)
4. else, return **DQElementEqIndex**($A[1 \dots (\lceil n/2 \rceil - 1)], offset$)

The initial call to this algorithm will be **DQElementEqIndex**($A[1 \dots n], 0$). The correctness of this algorithm relies on the observation that if $A[\lceil n/2 \rceil] < offset + \lceil n/2 \rceil$, then none of the $A[i]$ can be equal to i for all $i < \lceil n/2 \rceil$. This is because all the items are distinct and sorted. The same argument holds for the case when $A[\lceil n/2 \rceil] > \lceil n/2 \rceil + offset$.

The running time is $T(n) = T(n/2) + O(1)$, which by the Master Theorem is $O(\log n)$.

2.20 The $\Omega(n \log n)$ bound only applies to sorting algorithms that make ordering decisions by comparing pairs of items that are to be sorted. So, for us to beat this lower bound, we must create an algorithm that does not compare pairs of items. The sorting algorithm we will devise has a popular name called "Counting" sort. It works by first allocating an array of size M and counting how often each of the elements between $a = \min_i x_i$ and $b = \max_i x_i$ occurs. Here is the algorithm:

CountingSort($x[1 \dots n]$):

1. Let $a = \min x_i$, $b = \max x_i$ and $M = b - a$. Allocate an array $count[1 \dots M]$ of size M and initialize all entries to 0. Allocate a destination array where sorted values will be held, $sorted[1 \dots n]$. Set $index = 0$.
2. For $i = 1$ to n do: increment $count[x[i] - a + 1]$ by 1.
3. For $i = 1$ to M do:
 - (a) For $j = 1$ to $count[i]$ do: $sorted[index] = i + a$ and increment $index$ by 1.
4. Return $sorted$.

There are only $O(n)$ and $O(M)$ operations in the above algorithm making the entire runtime $O(n + M)$.

The correctness hinges on step 3, where after counting how often each item occurs we place the items in the sorted list in ascending order, repeating items when indicated by our counts.