Automata Theory (CSE4058-01)
Homework # ~~01~~ 04 – Solutions

3.19 We prove (3.20) by induction on m. The base case $m = 1$ is the familiar recursion, $F_\ell = F_{\ell-1} + F_{\ell-2}$. Now suppose we have established (3.20) for $m-1$, i.e.,

$$F_\ell = F_m F_{\ell-m+1} + F_{m-1} F_{\ell-m}.$$

But $F_{\ell-m+1} = F_{\ell-m} + F_{\ell-m-1}$, so

$$\begin{aligned} F_\ell &= F_m(F_{\ell-m} + F_{\ell-m-1}) + F_{m-1} F_{\ell-m} \\ &= (F_m + F_{m-1})F_{\ell-m} + F_m F_{\ell-m-1} \\ &= F_{m+1} F_{\ell-m} + F_m F_{\ell-m-1}. \end{aligned}$$

So, by induction, we have (3.20) for all $m \geq 1$. Setting $m = \ell$ then gives (3.21).

Now suppose we wish to compute $F_\ell \bmod p$. According to (3.21), we can use a divide-and-conquer approach. Let $T(\ell)$ be the time its takes to compute F_ℓ. But if we use (3.21) directly, and ignore the ± 1s, we get

$$T(2\ell) = 2T(\ell),$$

even if we ignore the time it takes to multiply and add the terms in (3.21). The solution to this is $T(\ell) = \Theta(\ell)$. But we want a running time which is polynomial in $n = \log_2 \ell$, not in ℓ.

The problem is that this direct recursion recomputes some Fibonacci numbers many times. A better approach is to compute F_ℓ and $F_{\ell+1}$ simultaneously. Using (3.21) we can reduce the problem of computing the pair $(F_{2\ell}, F_{2\ell+1})$ to that of computing $(F_\ell, F_{\ell+1})$ (using the fact that $F_{\ell-1} = F_{\ell+1} - F_\ell$), along with some addition and multiplication of n-bit integers. If the time it takes to do this for an n-bit ℓ is $f(n)$, then since we can multiply two n-bit integers in $O(n^2)$ time (or even less), we get the recurrence

$$f(n) = f(n-1) + O(n^2).$$

to which the solution is $T(n) = O(n^3)$.

3.22 Let's first find the length of the longest increasing subsequence, or l.i.s. for short, as opposed to finding the subsequence itself. We use dynamic programming. Let $f(i,t)$ be the length of the l.i.s. of $s_i, s_{i+1}, \ldots, s_n$ such that its first element is greater than t. Let's assume that the s_i are nonnegative, so that the l.i.s. of the entire list has length $f(1,0)$.

Now since s_i is either in the l.i.s. of $s_i, \ldots, s_n$ or not, and since it only counts towards $f(i,t)$ if $s_i > t$, we have

$$f(i,t) = \begin{cases} \max\Big(1 + f(i,s_i),\, f(i+1,t)\Big) & \text{if } s_i > t \\ f(i+1,t) & \text{if } s_i \leq t \end{cases}$$

If we calculate $f(1,0)$ recursively, we will ask about at most n^2 different values of $f(i,t)$, since $1 \leq i \leq n$ and t will always be one of the s_i. So while this recursive function would take exponential time if we recalculate $f(i,t)$ every time, if we memorize the values of $f(i,t)$ we have already calculated it will take $O(n^2)$ time.

We can modify the function f so that it returns the l.i.s. instead of its length. Just replace $1 + f(i,s_i)$ with $s_i \cdot f(i,s_i)$ where $\cdot$ denotes concatenation, and if a and b are strings let $\max(a,b)$ return the longer one.

3.30 We use dynamic programming. Recursively, we have $A \rightsquigarrow w$ either if there are nonempty strings u, v and variables B, C such that $w = uv$, $B \rightsquigarrow u$, $C \rightsquigarrow v$, and $(A \to BC) \in R$—or, in the base case, if w is a single symbol and $(A \to w) \in R$.

If we apply this recursive strategy, every subproblem we encounter will ask whether $A \rightsquigarrow u$ for some substring u of v. If $|w| = n$, there are $O(n^2)$ such substrings, so the number of different subproblems is $O(n^2|V|)$. Since V is given as part of the input, this is polynomial in the total size of the input.

3.47 One direction is easy. If there is a subset S on the left which is connected to fewer than $|S|$ vertices on the right, then by the pigeonhole principle there is no way to find partners for all the elements of S. Therefore if there is a perfect matching, every subset on the left is connected to a subset on the right which is at least at large.

In the other direction, suppose there is no perfect matching. Then according to the reduction of Figure 3.23, the MAX FLOW from s to t is less than n. By duality, the MIN CUT is also less than n, so there is some cut consisting of $c < n$ edges which separates s from t.

We claim that these edges might as well be among those leading out of s or into t in Figure 3.23, instead of in the middle layer of edges from the original bipartite graph. To see this, suppose the cut includes an edge (i, j) where i is one of the n vertices on the left and j is one of the n vertices on the right. This edge only blocks one path from s to t: namely, the path $s \to i \to j \to t$. If we replace (i, j) with (s, i) or (j, t), we still block this path, and perhaps others as well.

So, there is a cut consisting of x edges leading from s to some set of vertices on the left, and y edges leading from some set of vertices on the right to t, where $x + y < n$. Call these sets X and Y respectively, where $|X| = x$ and $|Y| = y$.

Now, for these edges to block all paths from s to t, there must be no edges from $\overline{X}$ to $\overline{Y}$. Thus if $S = \overline{X}$, the set T of vertices on the right that S is connected to is a subset of Y. But since $y < n - x$, we have $|T| \leq |Y| = y < n - x = |S|$. So, if there is no perfect matching, there is a subset on the left connected to a smaller subset on the right.