4041 Homework 4

Fletcher Gornick

October 24, 2021

21.3

21.3-3

Give a sequence of m `make-set`, `union`, and `find-set` operations, n of which are `make-set` operations, that takes $\Omega(m \lg n)$ time when we use union by rank only.

First, we will note that there are n elements, otherwise calling `make-set` n times is pointless, so we'll call these nodes $x_1, x_2, \ldots, x_n$. If we're going to introduce $\lg n$ complexity into our operations, the only way I can think to do this would be to use `union` to link these elements in a sort of binary tree structure. Let us also assume n is a power of two, because if it's not, then we would need to call `union` extra times which would only further worsen our complexity, meaning we would still be bounded by $\Omega(m \lg n)$.

Our sequence will first call `make-set` n times. Then we will call `union` $n/2$ times to link pairs of nodes, then call it another $n/4$ times to link the pairs into groups of 4. We will keep calling `union` until all the nodes are linked in a sort of binary tree structure. Assuming $n = 2^k$ for some k, the number of calls to union is $\frac{n}{2} + \frac{n}{4} + \cdots + \frac{n}{n} = n - 1$ (we actually know that any number of ways to call `union` is done in $n-1$ calls through the logic laid out in problem 15.2-6). There are k groups of these calls, each of which increases the tree's height by 1. So after we finish calling `union` $n-1$ times, we have a tree containing all the elements with a height of $k = \lg n$.

Finally we can call `find-set` m times which traverses the tree in $\lg n$ time for each call, and with m calls, we have a running time of $\Omega(m \lg n)$. This is obviously assuming that `find-set` doesn't implement path compression, otherwise the running-time would improve with each call.

21.3-4

Suppose that we wish to add the operation `print-set(x)`, which is given a node x and prints all the members of x's set, in any order. Show how we can add just a single attribute to each node in a disjoint-set forest so that `print-set(x)` takes time linear in the number of members of x's set and the asymptotic running times of the other operations are unchanged. Assume that we can print each member of the set in $O(1)$ time.

Disjoint set forests already have rank variables in each of their nodes, so we can introduce a new variable to make printing the whole set linear time. We can create a list member variable that contains each element in the set. The best data structure for this list would probably be a linked list, as that allows us to easily adjust the size.

So every time we call `make-set(x)`, it'll add a single node linked list containing element x, which is done in $O(1)$ time, so `make-set` keeps it's same run-time. Then every time we call `union(x,y)`, it'll take the node with the higher rank (or an arbitrary one if the rank is equal), and it'll become the new root, appending the other set's root-node list member variable onto it's own. So say `x.rank` $\geq$ `y.rank`, assuming x and y are the roots, then the list containing all the values in x will get linked with the list containing all the values in y, thus the resulting list contains all the elements in the new unioned set. This link is also an $O(1)$ procedure, so `union` run-time is unchanged. Finally, since `find-set` doesn't need any changes to help our `print-set` function, it's run-time is also unchanged.

So our `print-set(x)` function first calls `find-set(x)` to get the root node. We know that the worst case for `find-set` if we're not even using union by rank is $O(n)$, which most likely is not the case, but we will just say it's $O(n)$. Next, `print-set` just iterates through the linked-list member variable in the root node which is again $O(n)$. So we have `print-set(x)` is $O(n) + O(n) = O(n)$ time, and is thus linear.

15.2

15.2-2

Give a recursive algorithm `matrix-chain-multiply(A,s,i,j)` that actually performs the optimal matrix-chain multiplication, given the sequence of matrices $\langle A_1, A_2, \ldots, A_n \rangle$, the s table computed by `matrix-chain-order`, and the indices i and j. (The initial call would be `matrix-chain-multiply(A,s,1,n)`).

In this example, s represents the auxillary table where $s[i,j] = k$, k representing the optimal position to split the matrices. In other words, $A_iA_{i+1} \ldots A_j$ is optimal when split between A_k and A_{k+1}, making the new multiplication $(A_i \ldots A_k)(A_{k+1} \ldots A_j)$.

Knowing this, we can now get into what our algorithm should look like. First note that if $i = j$, then A_i and A_j are the same matrix, so we don't need to do a multiplication, and we can just return A_i. Also if $j = i + 1$ then we only have two matrices, so there's only one way to multiply them, and that's by simply returning $A_i \cdot A_j$.

Now is the actual recursive case. We want the algorithm to return $(A_i \ldots A_k)(A_{k+1} \ldots A_j)$, where k is denoted by $s[i,j]$, and thus $k + 1$ is defined as $s[i,j] + 1$. So our final case will recursively call `matrix-chain-multiply` from i to $s[i,j]$, and multiply it by the recursive call from $s[i,j] + 1$ to j, giving us an algorithm that looks like this...

```
matrix-chain-order(A,s,n)
    return matrix-chain-multiply-helper(A,s,1,n)

matrix-chain-multiply-helper(A,s,i,j)
    if (i == j) then return A_i
    if (i == j-1) then return A_i * A_j
    return mcmh(A,s,i,s[i,j]) * mcmh(A,s,s[i,j]+1,j)
```

PS: I abbreviated `matrix-chain-multiply-helper` to `mcmh` to make the algorithm look a little prettier.

15.2-3

Prove that $P(n) \geq \frac{1}{4}2^n$ by induction (*Hint:* Base cases are $n = 1, 2, 3$).

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \displaystyle\sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$$

Let's first look at the 3 base cases.

$$P(1) \;=\; 1 \;\geq\; \frac{1}{4}2^1$$

$$P(2) \;=\; \sum_{k=1}^{1} P(k)P(2-k) \;=\; P(1)P(1) \;=\; 1 \;\geq\; \frac{1}{4}2^2$$

$$P(3) \;=\; \sum_{k=1}^{2} P(k)P(3-k) \;=\; P(1)P(2) + P(2)P(1) \;=\; 1+1 \;=\; 2 \;\geq\; \frac{1}{4}2^3$$

Now, since we know $P(n)$ holds for $n = 1, 2, 3$, we can assume $P(k)$ is true for some $k > 1$, and show that $P(k+1)$ must be true. And by assuming $P(k)$ to be true, I mean $P(k) \geq \frac{1}{4}2^k$. We can proceed like so...

$$\begin{aligned} P(k+1) &= \sum_{l=1}^{k} P(l)P(k+1-l) \\ &= P(1)P(k) + P(2)P(k-1) + \cdots + P(k-1)P(2) + P(k)P(1) \\ &\geq \frac{1}{4}2^k + P(2)P(k-1) + \cdots + P(k-1)P(2) + \frac{1}{4}2^k \quad \text{(inductive hypothesis)} \\ &\geq \frac{1}{4}2^k + \frac{1}{4}2^k \\ &= \frac{1}{2}2^k \\ &= \frac{1}{4}2^{k+1} \end{aligned}$$

Therefore, by the principle of mathematical induction, the statement $P(n) \geq \frac{1}{4}2^n$ holds for all $n \in \mathbb{N}$.

15.2-6

Show that a full parenthesization of an n-element expression has exactly $n - 1$ pairs of parentheses.

The easiest way to show that the full parenthesization of an n-element expression consists of $n - 1$ pairs of parentheses is to use induction. So let us first look at the base case. When $n = 1$ there's one element in the expression, so no multiplication is needed and therefore, no parenthesization either. $n - 1 = 0$, so the base case holds. Let's also look at $n = 2$ even though it's not all that necessary, but with two matrices, they can just be multiplied together normally, but we must put parentheses around the two in order to make sure they get multiplied together first on an outer case, so $n = 2$ leads to 1 pair of parentheses, meaning the above statement still holds.

For our inductive step, we will use strong mathematical induction, so assuming the parenthesization of a k-element expression consists of $k - 1$ pairs of parentheses, AND that the parenthesization of all l-element expressions where $1 \leq l \leq k$ consists of $l - 1$ pairs of parentheses, we will show that the full parenthesization of an expression with $k + 1$ elements consist of k pairs of parentheses.

Let's say that the optimal split of our $k+1$-element expression $A_1 \cdot A_2 \cdots A_{k+1}$ is at A_l, this gives us $(A_1 \cdots A_l) \cdot (A_{l+1} \cdots A_{k+1})$ As the best spot to split our big expression. These two sub-expressions also have their own optimal parenthesization (including the parentheses I put on the outside of each of them). And by our inductive hypothesis, since both these sub-expressions have less than $k + 1$ elements, their optimal parenthesization is equal to the number of elements subtracted by 1. So the number of pairs of parentheses for $A_1 \cdots A_l$ is $l - 1$, and the number of parentheses for $A_{l+1} \cdots A_{k+1}$ is $(k + 1) - (l + 1) = k - l$. Now to find our how many pairs of parentheses we need for $k + 1$ elements, we must first add them, so we now have $l - 1 + k - l = k - 1$ pairs of parentheses. Finally we need one more pair to group our entire expression, giving us $k - 1 + 1 = k$ pairs of parentheses, therefore, the statement holds for our inductive step as well.

In conclusion, by the principle of strong mathematical induction, the statement that a full parenthesization of an n-element expression has exactly $n - 1$ pairs of parentheses holds for all $n \in \mathbb{N}$.

15.4

15.4-3

Give a memoized version of `lcs-length` that runs $O(mn)$ time.

Before writing the pseudocode, let's think about the cases first. Assuming we're taking in arguments X, Y which are the sequences, and i, j which is the index of the of our array `c[0...m, 0...n]`, which represents the longest substring of our sequences X_i and Y_j. So if $i = 0$ or $j = 0$ then we have a 0-length sequence, which means the longest common subsequence is 0. Now if $i \neq 0$ and $j \neq 0$, and $X_i = X_j$, then our longest common subsequence is the lcs of X_{i-1} and Y_{j-1} plus 1 for our new character. Finally if $X_i \neq Y_j$ then our lcs is either the lcs of X_{i-1}, Y_j, or the lcs of X_i, Y_{j-1}, we choose whichever one is bigger.

We can add memoization to this by initializing every value of `c[0...m, 0...n]` to negative infinity, so in our memoized algorithm, we first check if the value is positive, and if it is, then it's already been calculated, so we can return, so our algorithm looks like this...

```
lcs-length(X,Y)
   for i to X.length
      for j to Y.length
         c[i,j] = -inf
   memoized-lookup-index(X, Y, X.length, Y.length)

memoized-lookup-index(X,Y,i,j)
   if (c[i,j] >= 0) then return c[i,j]
   if (i == 0 or j == 0) then c[i,j] = 0
   else if (X_i == X_j) then c[i,j] = memoized-lookup-index(X,y,i-1,j-1)
   else
      a = memoized-lookup-index(X,y,i-1,j)
      b = memoized-lookup-index(X,y,i,j-1)
      c[i,j] = max(a,b)
   return c[i,j]
```

15.4-5

Give an $O(n^2)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers.

The easiest way to solve this problem without going through the whole dynamic programming process and finding a unique solution to this problem would be to utilize the longest common substring algorithm which was already provided to us in section 15.4.

First thing we do is copy the sequence into a new array, this way we can also keep track of the old one. Next, we can sort our newly created array. We can do this basically any way we want as long as it's $O(n^2)$ or better. So let's just say we're using quicksort which is $O(n \lg n)$ average case and $O(n^2)$ worst case, and for the sake of simplicity, let's just say it's $O(n^2)$. So now we have our original sequence of n numbers in it's original order, and a new sorted sequence, both are length n.

Now we can call the `lsc-length` function on our two sequences (we'll call them X and Y) which will output the arrays `b[1..n, 1...n]` and `c[0...n, 0...n]`. `c[i,j]` represents the length of the longest common subsequence for sequences X_i, Y_j. `b[i,j]` points to the table elemnent corresponding to the optimal subproblem, this is the array we will use to construct our optimal solution. This algorithm has a running time of $O(mn)$ where m and n are the length of the two sequences. Since the sequences we input both have length n, this algorithm runs in $O(n^2)$ time.

Finally, we can call `print-lcs` which takes in our auxillary table b and prints out the subsequence we solved for. This operation is done in $O(m + n)$ time. Again, since the two sequences are length n, this algorithm has a $O(2n) = O(n)$ running time.

Putting it all together, we have our array copy with $O(n)$ running time, then a sort, which is $O(n^2)$ or $O(n \lg n)$ if we're talking average (or if we just use merge sort or heap sort). Next is the algorithm that actually does the work, which is `lsc-length`, with an $O(n^2)$ running time. And finally is our `print-lcs` function with $O(n)$ running time. This gives us a new time complexity of $O(n) + O(n^2) + O(n^2) + O(n) = O(n^2)$. This algorithm produces the longest monotonically increasing subsequence because if you take a sequence, and it's corresponding sorted sequence, the `lcs` procedure must produce the longest common substring, and it must be increasing because our second sequence is in increasing order.