Pset 4

Darwin Do

March 8, 2022

(a) Darwin Do

(b) 919941748

(c) Collaborators: Raffael Davila, Graham Stodolski

(d) I have followed the academic integrity and collaboration policy

(e) Hours: 16

1 Finding Element

(a) We can solve this problem using binary search. We split our input of size n into 2 groups of size $n/2$ with a "partition" element dividing between the two groups. If the partition element equals the target, we return the index of the partition. If the partition is less than the target, we recurse on the right group. Otherwise, the partition is greater than the target and we recurse on the left group.

This algorithm works as it relies on the fact that successive elements can only change by 1 and $p \leq t \leq q$. Let's define the partition element as x. If $x \leq t$, then $x \leq q$ and t has to appear somewhere in between x and q as every value between x and q will be reached as we increase by 1 to get to q. The same logic applies for the $x \geq t$ case, t would have to appear between p and x.

(b) $\star$ Note that I'm using array inclusive array range notation (i.e. $A[1:x]$) means all elements in the array A from the first element to element x, including element x.

function FindElement(A, t)
  $n =$ length(A)
  $x = A[\lceil \frac{n}{2} \rceil]$
  if $x == t$ **then return** $\frac{n}{2}$
  else if $x > t$ **then return** FindElement($A[1 : \frac{n}{2} - 1], t$)
  else
   return FindElement($A[\frac{n}{2} + 1 : n], t$)

(c) $T(n) = T(\frac{n}{2}) + 1$

(d)

$$A = 1, B = 2, D = 0$$
$$R = \frac{A}{B^D} = \frac{1}{2^0} = 1$$

So according to the Master Theorem: $T(n) = O(n^D \log n) = O(\log n)$

2 Maximum Sum

(a) We use a divide and conquer algorithm that divides each input into 2 parts with size $n/2$. The algorithm keeps doing this until we reach the base case where $n = 1$. In this case, we simply return the value of the element. Otherwise, for the "conquering" step of the algorithm, we let L equal the solution for the left part and R equal the solution for the right part as returned by our recursive calls. We also define LS as the largest sum of the left part starting from $A[\lceil n/2 \rceil]$ to $A[0]$ and RS as the largest sum of the right part starting from $A[\lceil n/2 \rceil + 1]$ to $A[n]$. We then return the maximum of the following options, L, R, or $LS + RS$.

This algorithm works as each recursive call will always return the biggest contiguous sum within its input, whether that be the sum from the left side, the sum from the right side, or a sum that spans between the two sides.

(b)
function MaxSum(A)
  $n =$ length(A)
  if $n == 1$ **then return** $A[1]$
  $M = \lceil \frac{n}{2} \rceil$
  $L =$ MaxSum($A[1 : M]$)
  $R =$ MaxSum($A[M + 1 : n]$)
  $LS = 0; RS = 0;$
  $tempTotal = 0$
  for i from M to 1 **do**
    $tempTotal = tempTotal + A[i]$
    if $tempTotal > LS$ **then**
      $LS = tempTotal$
  $tempTotal = 0$
  for i from $M + 1$ to n **do**
    $tempTotal = tempTotal + A[i]$
    if $tempTotal > RS$ **then**
      $RS = tempTotal$
  return max($L, R, LS + RS$)

(c) $T(n) = 2T(\frac{n}{2}) + O(n)$

(d)

$$A = 2, B = 2, D = 1$$
$$R = \frac{A}{B^D} = \frac{2}{2^1} = 1$$

So according to the Master Theorem: $T(n) = O(n^D \log n) = O(n \log n)$

3 Longest Common Substring

(a) $T(i, j)$ is the length of the largest common substring that ends at index i in x and index j in y. The final answer of the algorithm will be the max value of all table entries.

(b)

$$T(i,j) = \begin{cases} T(i-1, j-1) + 1 & x[i] = y[j] \\ 0 & x[i] \neq y[j] \end{cases}$$

The base cases are the indexes of the recurrence before the first character. In this homework with 1-indexing, our base cases would be $T(0, j) = 0$ and $T(i, 0) = 0$.

The recurrence works as each value in the table holds the value of the longest substring that ends at index i in string x and index j in string y. If this common substring with length L can grow, then $x[i+1] = y[j+1]$. We know then that the current common substring now has length $L' = L + 1$. Otherwise, if the next characters are not equal, then the common substring has ended and the length of the common substring *that ends at indexes* $\boldsymbol{i}$ *and* $\boldsymbol{j}$ *in strings* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *respectively* is 0. The piecewise function of the recurrence accounts for this.

(c)
```
function LongestCommonSubstring(x, y)
    maxLength = 0
    for all i from 0 to n do
        for all j from 0 to m do
            if i == 0 or j == 0 then
                sub[i][j] = 0
            else if x[i] == y[j] then
                sub[i][j] = sub[i − 1][j − 1] + 1
                if sub[i][j] > maxLength then maxLength = sub[i][j]
            else
                sub[i][j] = 0
    return maxLength
```

(d) The runtime is $O(nm)$. We iterate through all n characters in string x and for every character, we look at every character in string y of length m. There is a constant amount of work for each inner iteration as we are simply updating an array with a previously supplied value. Therefore the runtime is $O(nm)$ total.

4 Infinitely Many Rods

(a) $T(i) =$ `True` if a rod of length i can be assembled with the given subrods and $T(i) =$ `False` otherwise. The final answer of the algorithm will be the value stored at $T(S)$

(b)

$$T(i) = \begin{cases} \texttt{True} & \forall(1 \leq x \leq n)(\exists \ell_x : i - \ell_x > 0 \wedge T(i - \ell_x) = \texttt{True}) \\ \texttt{False} & \forall(1 \leq x \leq n)(\nexists \ell_x : i - \ell_x > 0 \wedge T(i - \ell_x) = \texttt{True}) \end{cases}$$

The base case is $T(0) =$ `True`

The recurrence is correct as at index i with all the recurrences below i already solved, we know that we only need to add a minimum of one rod from one of the previously solved solutions to form a rod of length i if it exists. Therefore we cycle through all the availble rod sizes ℓ_x and see if the recurrence at $i - \ell_x$ is true. If it is, then we know we are able to add a rod of size ℓ_x to form a rod of size i and $T(i)$ is true. Otherwise if none of the supplied rods work, we know that it is not possible to form a rod of size i with our given inventory.

The base case is $T(0) =$ `True` as we want singleton rods to be true (i.e. a rod of size 3 can be formed by a single rod of length 3). This also intuitively makes sense as it doesn't matter what length of rods we have in order to make "no" rod.

(c)

```
function CheckValidRod(ℓ_1...ℓ_n, S)
    rod[0] = True
    for all s from 1 to S do
        for all i from 1 to n do
            if s − ℓ_i >= 0 and rod[s − ℓ_i] == True then
                rod[s] = True
                break
            else
                rod[s] = False
    return rod[S]
```

(d) The runtime is $O(nS)$. In our dynamic programming algorithm, we update our table of solutions from $rod[1]$ to $rod[S]$. Each table update takes $O(n)$ time as the worst case scenario is to cycle through all rods supplied. Therefore the total runtime is $O(nS)$

5 At Most One Rod

(a) $T(i, j) =$ `True` if a rod of length j can be assembled with the rods before ℓ_i including OR NOT including rod ℓ_i. $T(i, j) =$ `False` otherwise. The final answer will be the value of $T(S, n)$.

(b) $T(i, j) = T(i-1, j) =$ `True` $\vee\, T(i-1, j-\ell_i) =$ `True`

The base cases are $T(0, 0) =$ `True` and $T(0, 1...S) =$ `False`

The recurrence is composed of two parts. By checking if $T(i-1, j)$ is true, we are seeing whether length j has already been reached by the previously tested rods for length j. If it has, there is no need to factor in rod ℓ_i, we already know it is possible to construct j. Therefore the value of $T(i, j)$ should be true. On the otherhand, if we haven't reached j yet, we want to see if adding rod ℓ_i will get us to j. If this is true, then it must be possible to reach length $j - \ell_i$ with the previous rods. To check this, we need to look at $T(i-1, j-\ell_i)$. If this value is true, then that value is possible and we know we can construct length j with rods up to ℓ_i.

(c)
```
function CheckValidRod(ℓ_1...ℓ_n, S)
    rod[0][0] = True
    for all i from 0 to n do
        for all j from 0 to S do
            if i == 0 and j ≥ 1 then
                rod[i][j] = False
            else if i > 0 then
                rod[i, j] = rod[i−1, j] == True ∨ rod[i−1, j−ℓ_i] == True
    return rod[n, S]
```

(d) The runtime is $O(nS)$ The runtime is $O(nS)$ as we have a nested for loop iterating through S and n. Each root operation of the nested for loops is to update the *rod* table which takes constant time as we are simply comparing two values in the array that have already been computed.