2019 ADA HW 1

b07902064 資工二 蔡銘軒

October 16, 2019

Problem 5

(1) Asymptotic Notations

(a) False. Let $f(n) = n$ and $g(n) = n^2$. Then we have $n^2 + n \neq O(n)$

(b) True.

(c) True.

(d) False. Let $f(n) = 2^n$. We have to show $\exists c_1, c_2, n_0 : \forall n \geq n_0, 0 \leq c_1 \cdot 2^{\frac{n}{2}} \leq 2^n \leq c_2 \cdot 2^{\frac{n}{2}}$, which implies $c_2 \geq 2^{\frac{n}{2}}$. This makes it impossible for c_2 to be a constant.

(e) False. $log_2(n!) \leq log_2(n^n) = nlog_2 n = O(nlgn) \;\neq \Omega(n^2)$

(2) Solve Recurrences

(a) By master theorem, $a = 6$, and $b = 3$, we have $n^{log_b a} = n^{log_3 6} \approx n^{1.63}$.
Let $\epsilon \in (0, \log_3 6 - 1]$, then the form $f(n) = O(n^{\log_b a - \epsilon})$ is satisfied. So we conclude that $T(n) = \Theta(n^{\log_3 6})$ by master theorem.

(b) First we observe that $T(n) = T(\frac{n}{3}) + T(\frac{n}{4}) + T(\frac{n}{12}) + 24n \geq T(\frac{n}{12}) + T(\frac{n}{12}) + T(\frac{n}{12}) + 24n = 3 \cdot T(\frac{n}{12}) + 24n$, which is $\Theta(n)$ by master theorem. Therefore, we have $T(n) = \Omega(n)$.
With the help of recursion tree, which has $24n$ at the first level, $16n$ at the second level, $\frac{32}{3}n$ at the third level, and so on, we can see that the tree will eventually reach the base case and stop growing; and at some level, the tree will stop to follow the growing pattern as some branches arrive at the base case ealier. So we have an upper bound $24n + 24 \cdot \frac{2}{3}n + 24 \cdot (\frac{2}{3})^2 n + ... = 24 \cdot \frac{1}{1-\frac{2}{3}} n = 72n$, which gives $T(n) = O(n)$.
Now that we have $T(n) = \Omega(n)$ and $T(n) = O(n)$, we can say $T(n) = \Theta(n)$.

(c) First we apply the transformation $S(n) = \frac{T(n)}{n}$ and arrive at $S(n) = S(\sqrt{n}) + 2\lg(n)$
Let $k = \lg(n)$, we have $S(2^k) = S(2^{\frac{k}{2}}) + 2k$. Further let $F(k) = S(2^k)$, then we have $F(k) = F(\frac{k}{2}) + 2k$.
By master theorem, $F(k) = \Theta(k)$. Changing back from $F(k)$ to $S(k)$, we have $S(k) = \Theta(\lg(k))$.
Finally, substituting back to $T(n)$, we obtain $T(n) = \Theta(n \lg n)$.

(d) $T(n) = 2 \cdot T(\frac{n}{2}) + \frac{4n}{\lg n}$
$= 2 \cdot (2 \cdot T(\frac{n}{4}) + \frac{2n}{\lg \frac{n}{2}}) + \frac{4n}{\lg n}$
$\vdots$
$= 2^k \cdot T(\frac{n}{2^k}) + 4n \sum_{i=0}^{k-1} \frac{1}{\lg \frac{n}{2^i}}$
$= 2^k \cdot T(\frac{n}{2^k}) + 4n \sum_{i=0}^{k-1} \frac{1}{\lg n - i}$
$\approx n + 4n \cdot \lg\lg n = \Theta(n \lg\lg n)$ (By the inequalities $n \lg\lg n \leq n + 4n \lg\lg n \leq 10^9 n \lg\lg n$ for $n \geq 3$)

Problem 6

(1) The algorithm is based on merge sort. When merging the left and right arrays, the elements in the right array has index larger than those in the left array. So if some element r_j in the right array comes before $l_i, l_{i+1}, \ldots, lmid$, then each of these element can form an inversion with r_j. With the idea in mind, we can devise the following algorithm:

```
merge(Arr, left, mid, right):
   Creat a temporary Array tmp
   left_pointer = left
   right_pointer = mid + 1
   inversion_cnt = 0
   while left_pointer != mid + 1 && right_pointer != right + 1:
      if Arr[left_pointer] < Arr[right_pointer]:
         put Arr[left_pointer] into tmp
         left_pointer += 1
      else if Arr[right_pointer] < Arr[left_pointer]:
         inversion_cnt += (mid - left_pointer + 1)
         put Arr[right_pointer] into tmp
         right_pointer += 1
   while left_pointer != mid + 1:
      put Arr[left_pointer] into tmp
      left_pointer += 1
   while right_pointer != right + 1:
      put Arr[right_pointer] into tmp
      right_pointer += 1
   now tmp has stored the elements in Arr[left...right] in sorted order
   Copy tmp to Arr[left...right] so that Arr[left...right] is sorted
   return inversion_cnt

Inversion(Arr, left, right):
   if left == right:
      return 0
   mid = (left + right) / 2
   left_cnt = Inversion(Arr, left, mid)
   right_cnt = Inversion(Arr, mid + 1, right)
   merge_cnt = merge(Arr, left, mid, right)
   return left_cnt + right_cnt + merge_cnt
```

(2) We first look at the "merge" function. Comparing two numbers takes $O(1)$ time, and there are at most n comparison, where n is the total number of elements in the two arrays. Copying the elements to and from the temporary array also takes $O(n)$ time. So we conclude that the merge function runs in $O(n)$ time.
Then we look at the "Inversion" function. Let the running time of the function be $T(n)$, then we have:
Base case: $O(1)$
Recursive case: $2 \cdot T(\frac{n}{2}) + O(n)$. The $O(n)$ is due to the "merge" function.
We obtain:

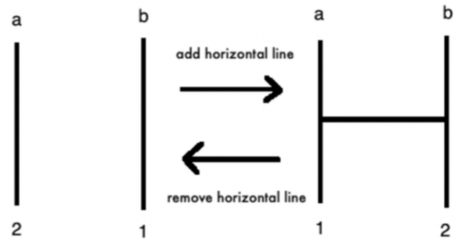
$$T(n) = \begin{cases} O(1) & if\ n \leq 1 \\ 2T\left(\frac{n}{2}\right) + O(n) & else \end{cases}$$

By master theorem, the overall complexity is $\Theta(n \lg n)$, which also means $O(n \lg n)$.

(3) The operation bubble sort performs is swapping two adjacent elements. Bubble sort terminates when the inversion count is 0, which means the sequence is sorted. We show that every swap reduces the

inversion count by exactly one, so the number of exchanges is equal to the number of inversions.
Suppose a_i and a_{i+1} are elements in S such that $a_i > a_{i+1}$. Consider swapping the two elements. For elements a_j such that $j = 0, 1, ..., i-1$, their positions relative to a_i and a_{i+1} stay the same after the swap, so the inversion count contributed by a_j and a_i, a_j and a_{i+1} doesn't change. Similarly, for a_j such that $j = i+2, i+3, ..., |S|$, their positions relative to a_i and a_{i+1} remain after the swap, so the inversion count also remains. Now wee see every swap removes only the inversion contributed by a_i and a_{i+1}. Thus the overall inversion count is reduced by one by every swap, which means the number of exchanges needed in bubble sort is the number of inversions.

(4) If there are exactly N constraints, either the constraints contradict each other, or each player is assigned a particular destination. Here we only discuss the latter, as the former is unsolvable.
We first recognize a horizontal line is equivalent to swapping two adjacent elements in an array, as demonstrated in the figure below.



We denote the initial state as a goes to 1, b goes to 2, etc, and the final state as the given constraints. However, we can reverse the order as removing a horizontal line to erase the swap is equivalent to adding an exactly same line again to cancel out the former line, e.g. swap the elements, and swap them again is the same as not swapping at all. So we can start with the final state, e.g. the given constraints, and check how many horizontal lines are required to get to the initial state. The problem is transformed to ***inversion count in an array*** as in (1).

```
horizontal_lines():
   Assign the constraints to each player. if contradiction is found, return an error
   Create an empty array A[] and assign values as follows:
      A[0] is the destination for a,
      A[1] is the destination for b,
      ...
   result = Inversion(A, 0, N - 1) // the funtion devised in (1)
   return result
```

The complexity is $O(n)$ for assigning constraints and creating the array plus $O(n \lg n)$ for calculating the number of inversions, so the overall complexity is $O(n) + O(n \lg n) = O(n \lg n)$

(5) First, we introduce the array $cons[]$ where $cons[i]$ is the ith player's destination.
In the previous problem, we have shown that the minimum horizontal lines required is equal to the number of inversions in array A, where A consists of elements $A[i] = cons[i]$. Now we fill the array $cons[]$ with the given constraints. If $|constraints| = N$, the problem is same as (4); if $|constraints| < N$, we claim that for every pair of players (i, j) who haven't been assigned a destination, we have $cons[i] < cons[j]$ if and only if $i < j$. The proof is as follows:
Assume the claim is satisfied, so we have $i < j$ in the following illustration:

$$\underbrace{\underbrace{\ldots\ldots}_{a}\, i\, \underbrace{\ldots\ldots}_{b}\, j\, \underbrace{\ldots\ldots}_{c}}_{A[]}$$

The inversion count contributed by i and j is given by:

$$(a_k, i)\ for\ a_k \in a\ and\ a_k > i \tag{1a}$$
$$(a_k, j)\ for\ a_k \in a\ and\ a_k > j \tag{1b}$$
$$(b_k, i)\ for\ b_k \in b\ and\ b_k < i \tag{1c}$$
$$(b_k, j)\ for\ b_k \in b\ and\ b_k > j \tag{1d}$$
$$(c_k, i)\ for\ c_k \in c\ and\ c_k < i \tag{1e}$$
$$(c_k, j)\ for\ c_k \in c\ and\ c_k < j \tag{1f}$$

Consider swapping i and j, then the inversion count contributed by i and j is then:

$$(a_k, i)\ for\ a_k \in a\ and\ a_k > i \tag{2a}$$
$$(a_k, j)\ for\ a_k \in a\ and\ a_k > j \tag{2b}$$
$$(b_k, i)\ for\ b_k \in b\ and\ b_k > i \tag{2c}$$
$$(b_k, j)\ for\ b_k \in b\ and\ b_k < j \tag{2d}$$
$$(c_k, i)\ for\ c_k \in c\ and\ c_k < i \tag{2e}$$
$$(c_k, j)\ for\ c_k \in c\ and\ c_k < j \tag{2f}$$
$$(i, j) \tag{2g}$$

Clearly, the only difference is between $(1c) + (1d)$ and $(2c) + (2d) + 1$. Since we have $i < j$, it is obvious the latter is bigger than the former, and thus the latter has more inversions.
Now that the claim has been proven, we can follow the similar idea in (4) to solve this problem:

```
horizontal_line():
   Create the array const[] and fill the constraints.
   if contradiction is found:
      return an error
   else:
      fill the empty cells with unused numbers in increasing order
   Construct array A[] such that A[i] = const[i]
   result = Inversion(A, 0, N - 1)
   return result
```

Every operation before the "Inversion" function can be done in $O(n)$ complexity, so the overall complexity is $O(n) + O(n \lg n) = O(n \lg n)$

Problem 7

(1) Let L be the length of the block, d_1 be the distance to the left end, and d_2 be the distance to the right end. For the block to unfold, we must have $d_1 = k_1 L$ and $d_2 = k_2 L$ for some $k_1, k_2 \in \mathbb{N} \cup \{0\}$, and $\frac{N}{L} = 2^i$ for some $i \geq 0$. The first condition can be verified by modular arithmetic, and the second condition can be verified by checking whether $x\&(-x)$ equals 0. All these can be done in constant time.

(2) Assume the puzzle is solvable and we reduce the length of block to 1 by dividing L. We use a method similar to binary search. First we split the board in half and check whether the block is on the left side or on the right side. If it's on the left side, we recursively fill the left side and unfold to right, filling the right side in one operation and vice versa. The complexity is $O(\lg n)$ for binary search.

```
puzzle(Arr, left, right, block):
   if reach the block: // base case
      return
   mid = (left + right) / 2
   if block is on the left:
      puzzle(Arr, left, mid, block)
      Arr <- right // after filling the left, we unfold to the right
   else:
      puzzle(Arr, mid + 1, right)
      Arr <- left // after filling the right, we unfold to the left

main():
   Create Arr[] to store the sequence of operation
   puzzle(Arr, 1, N, block) // block stands for the position of the block
   Output Arr[]
```

(3) First we note that $O(d_1 + d_2)$ is the same as $O(N)$. If we record the unfolding process in a sequence and each entry can be either L or R, meaning unfold to the left and unfold to the right, respectively, then it is obvious the sequence has length at most $\lg N$ when the puzzle is filled or stepped over bound. Since every entry has 2 options, and there are $\lg N$ entries, the number of status is $2^{\lg N}$. Then we consider the possible status after unfolding 0 time, 1 time, 2 times, etc. We have

$$\sum_{i=0}^{\lceil \lg n \rceil} 2^i \leq 2^{\lg n+2} - 1 \leq 2^{\lg n+3} = 8n = O(n)$$

(4) Let the left end point be position 1, and we create an boolean array $dp[]$ where $dp[i]$ indicates whether we can fill positions from 1 to i.
As base case, we fill $dp[0]$ as *true*. For every block b_i, we enumerate all its possible unfolding operations, and if the left end point is connected with some cell marked *true* and the right end does not cross the boundary, we know it is possible to fill the board from position 1 to the current right end point, so we mark $dp[\textit{right end point}]$ as *true*. After we do this to every block, we check if $dp[N]$ is *true* or not.

```
puzzle():
   Create array dp[] (initialized to false) to store answer, block[] to store the
       information of blocks
   dp[0] = true // the first cell of the board is at position 1, and we set position 0 as
       the base case
   for every block[i]:
      for every possible status reachable by unfolding block[i]:
         //the left end is at position L, and the right end is at position R
         //by possible status we mean L is between block[i - 1] and block[i], R is between
             block[i] and block[i + 1] since we cannot step over bound
         if dp[L - 1] == true:
            dp[R] = true
   return dp[N]
```

(5) If we view from a recursive perspective, then we have:
overlapping subproblem: if position $[i, j_1]$, $[i, j_2]$, $[i, j_3]$, ... can be filled, then each of them will ask whether $[1, i-1]$ can be filled. So $[1, i-1]$ is an overlapping subproblem.
optimal substructure: $dp[i]$ is *true* if and only if there exists some j such that $j < i$ and $dp[j]$ is *true* and $[j+1, i]$ can be filled for some block.

(6) The creation of $dp[]$, reading input, and the enumeration of every possible status generated by every block give $\Omega(N)$.

If we let d_1 be the distance from the first block to the left end point, d_i for $2 \leq i \leq n-1$ be the distance between $block_{i-1}$ and $block_i$, and d_n be the distance from the last block to the right end point, then by conclusion in (3), we know all possible status we enumerate will not exceed $O(d_1+d_2)+O(d_2+d_3)+\cdots+O(d_{n-2}+d_{n-1})+O(d_{n-1}+d_n) = O(d_1+2d_2+2d_3+\cdots+2d_{n-1}+d_n) = O(2N) = O(N)$. Now we have $\Omega(N)$ and $O(N)$, we can say that our algorithm runs in $\Theta(N)$.

Reference

- b07902026 陳玉恆
- b07902028 林鶴哲
- b07902075 林楷恩
- b07902132 陳威翰