ADA HW #1 - Hand-Written

Student Name: 林楷恩
Student ID: b07902075

Problem 5 - Time Complexity & Recurrence

(1) (a) **False**, ex. $f(n) = n^2, g(n) = n$, $f(n) + g(n) \neq O(min(f(n), g(n))) = O(n)$

(b) **True**

(c) **True**

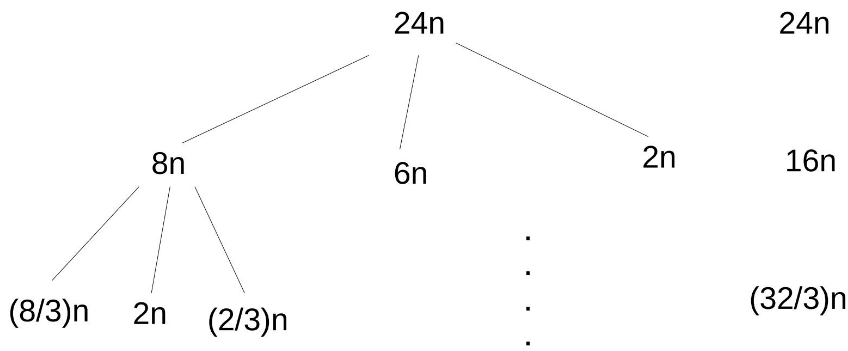
(d) **False**, ex. $f(n) = 2^{2n} \neq \Theta(f(n/2)) = \Theta(2^n)$

(e) **False**, proof:
$log_2(n!) = log_2(n) + log_2(n-1) + \cdots + log_2(1) \leq log_2(n) + log_2(n) + \cdots + log_2(n) = nlog_2(n)$, $cnlog(n) < n^2$ when n is large enough no matter how big constant c is. Thus we can never find a c and a n_0 s.t. $0 \leq cn^2 \leq nlog(n)$ for all $n \geq n_0$.

(2) (a) Apply *Master Theorem* with $a = 6, b = 3, f(n) = 108n$.
Take $\epsilon = 1, c = 1000, n_0 = 1 \implies 0 \leq f(n) \leq c\, n^{log_3 6 - \epsilon}$, $\forall n \geq n_0$, $f(n) = O(n^{log_3 6 - \epsilon})$
$\implies T(n) = \Theta(n^{log_3 6})$

(b) 1° $T(n) = T(\frac{n}{3}) + T(\frac{n}{4}) + T(\frac{n}{12}) + 24n > 24n$ $(\because T(n) \geq 1)$ ...①
$\forall\ n \geq 1,\ 0 \leq 24n < T(n) \implies T(n) = \Omega(n)$

2° Apply recursion-tree method:



the sum of all layers forms a geometric sequence, the upper bound of their sum is:
$\frac{24n}{1-\frac{2}{3}} = 72n \implies T(n) \leq 72n$
$\forall\ n \geq 1,\ 0 \leq T(n) \leq 72n \leq 100n \implies T(n) = O(n)$ ...②

By ① and ②, $T(n) = \Theta(n)$

(c)

$$Let\ S(n) = \frac{T(n)}{n},\ \frac{T(n)}{n} = \frac{T(\sqrt{n})}{\sqrt{n}} + 2lg\,n \implies S(n) = S(\sqrt{n}) + 2lg\,n$$

$$Let\ k = lg\,n,\ n = 2^k, S(2^k) = S(2^{k/2}) + 2k$$

$$Let\ F(k) = S(2^k),\ F(k) = F(k/2) + 2k$$

$$expand\ F(k) \implies F(k) = F(k/4) + k + 2k = \cdots = \frac{2k(1-(\frac{1}{2})^{log_2(k)})}{1-\frac{1}{2}} = 4k - 4k(\frac{1}{2})^{log_2(k)} = \Theta(k)$$

$$S(2^k) = \Theta(k) \implies S(n) = \Theta(lg\,n) \implies T(n) = n \times \Theta(lg\,n) = \Theta(n\,lg\,n)$$

(d)

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + \frac{4n}{lg\,n} \\
&= 2(2T(\frac{n}{4}) + \frac{2n}{lg\frac{n}{2}} = 4T(\frac{n}{4}) + \frac{4n}{lg\frac{n}{2}} + \frac{4n}{lg\,n} \\
&= 4(2T(\frac{n}{8}) + \frac{n}{lg\frac{n}{4}}) + \frac{4n}{lg\frac{n}{2}} + \frac{4n}{lg\,n} = 8T(\frac{n}{8}) + \frac{4n}{lg\frac{n}{4}} + \frac{4n}{lg\frac{n}{2}} + \frac{4n}{lg\,n} \\
&\vdots \\
&= 2^{lg\,n}T(1) + 4n(\sum_{k=1}^{lg\,n} \frac{1}{lg\frac{n}{2^k}}) \\
&= n + 4n(\sum_{k=1}^{lg\,n} \frac{1}{lg\,n - k}) \\
&= n + 4n(\sum_{k=1}^{lg\,n} \frac{1}{k}) \\
&= n + 4n\,ln\,lg\,n
\end{aligned}
$$

$$\forall\ n \geq 2,\ 0 \leq 1(n\,lg\,lg\,n) \leq n + 4n\,ln\,lg\,n \leq 10(n\,lg\,lg\,n)$$

$$T(n) = \Theta(n\,lg\,lg\,n)$$

Problem 6 - Controllable Ghost Leg

Inversion

(1) I apply a divide and conquer algorithm similar to *merge sort*. To solve the number of inversion in a range $[L, R]$, I first divide the interval into left and right subarray by the midpoint and recursively conquer them. Then, I need to consider those inversion which formed by two elements in different subarray. To achieve this in $\Theta(n)$, I make the current interval sorted before returning to its parent interval. So after the divide and conquer step finished, I have the left and right subarray sorted. For every element B[l] on the left, I consider the number of elements on the right smaller than B[l] (form an inversion). Because the right subarray is sorted, for a B[l] I can find an index p such that all elements in [midpoint+1, p] are smaller than B[l]. Furthermore, since the left subarray is also sorted, if I consider them from the small ones to the big ones, I can have the good property that p never decreases since the number of elements smaller than B[l] will not decrease if B[l] is getting larger.

Count the Number of Inversion

```
input: an unsorted and unique sequence B and a integer n = |B|
output: int -> the number inversion

// Call CountInversion(B, 0, n-1) to get the answer,
// where L and R is the current lower and upper bound of array
function CountInversion(B, L, R):
    // Base case ...Θ(1)
    if L = R:
        return 0
    // divide B into two subarray: [L, M] and [M+1, R] ...Θ(1)
    M ← ⌊ (L + R) ÷ 2 ⌋
    // conquer ...2T(n/2)
    ans ← CountInversion(B, L, M) + CountInversion(B, M+1, R)

    // combine ...Θ(n)
    p ← M + 1
    foreach i from L to M:
        while p ≤ R and B[p] < B[i]:
            p ← p + 1
        ans ← ans + (p − M − 1)

    // merge two sorted subarray into one ...Θ(n)
    tmp ← []
    p1 ← L
    p2 ← M + 1
    while p1 <= M and p2 <= R:
        if B[p1] < B[p2]:
            append B[p1] to tmp
            p1 ← p1 + 1
        else:
            append B[p2] to tmp
            p2 ← p2 + 1

    if p1 <= M:
        append B[p1...M] to tmp
    else if p2 <= R:
        append B[p2...R] to tmp

    B[L...R] ← tmp

    return ans
```

(2)
- **base case** is $\Theta(1)$, which is trivial.
- **the *combine* step**: we have two iterators i and p, i in outer-loop run from L to M which go through $\frac{n}{2}$ element and thus is $\Theta(n)$, while p in inner-loop run from M+1 to at most R plus the fact that p never decreases, so it is also $\Theta(n)$.
- **merge two sorted sequence into one sorted sequence** is also $\Theta(n)$, because the iterators p1 and p2 start from the left most index of the two subarray and they never decrease, so they go through each element in $[L...R]$ only and exactly once.

$$T(n) = \begin{cases} \Theta(1) & \ldots \textit{base case} \\ 2T(n/2) + \Theta(n) & \ldots n > 1 \end{cases}$$

Apply *Master Theorem*, with $a = 2, b = 2, f(n) = \Theta(n)$, and $f(n) = \Theta(n^{log_a b}) = \Theta(n)$. Thus, the time complexity of this algorithm is $O(n\,lg\,n)$

(3) To prove this, I first prove that the following claim is true:

- **Claim:** Each exchange when performing bubble sort makes the number of inversion of a sequence S decreases by exactly one.
- **Proof:** When performing bubble sort, exchanges only happen when the program detects an index i such that $S_i > S_{i+1}$. Because the two elements are adjacent, exchanging them does not affect the relative position for elements in $[1, i-1] \cup [i+2, n]$. That is, for elements in $[1, i-1]$, S_i and S_{i+1} still stay on their right, while for those in $[i+2, n]$, S_i and S_{i+1} still stay on their left, too. Thus, only the change on the relation between S_i and S_{i+1} needs to be considered. Before exchange, $S_i > S_{i+1}$ which forms a inversion. After exchange, $S_i < S_{i+1}$, which is not a inversion. So the number of inversion exactly decreases by one because of the exchange.

Because bubble sort keep exchanging elements until there is no inversion in S(i.e. sorted), and by claim each exchange reduces the number of inversioin by 1. Therefore, the number of inversions is equal to the number of exchanges when performing bubble sort.

Controllable Ghost Leg

(4) We can transform Controllable Ghost Leg Problem into Inversion Problem by the following algorithm, which constructs an array A to be the input of above `CountInversion` function:

Let the starting points(people) and the end points(prizes) labeled with 1 to N from left to right. Then, for every constraint mapping people i to prize j, we make A[i] = j. Since $|constraints| = N$, A is filled by exactly N numbers. The reason is that we want the element in position i end up going to position j and we just place j on the position i, we make the desired final state be a sorted array.

On the other hand, every horizontal line connecting two vertical lines x and y can be regarded as "exchanging the numbers on $A[x]$ and $A[y]$", because when players on x and y encounter a horizontal line, players on x should go to y while the one on y should go to x. The exchanging operations are performed in the order of the vertical positions of the horizontal lines placed from top to bottom.

As the result, the Ghost Leg Problem is equal to find the minimum number of exchanges on two adjacent elements to make an array sorted, which is the number of exchanges when performing bubble sort. Moreover, We have proved that it is equal to the number of inversion in an array. Therefore, we can just run `CountInversion` on A to get the answer.

To construct A, the algorithm go through N constraints, which is $O(N)$, while the `CountInversion` is $O(N\,lg\,N)$ Thus the overall time complexity is $O(N) + O(Nlg\,N) = O(N\,lg\,N)$

(5) The main idea is the same as previous problem, but in this case we have to deal with the starting points and prizes not assigned any constraints. The solution is: we iterate from 1 to N, if we find an "empty" position (A[i] has not yet been assigned), we greedily assigned it the minimum available prize index to it. To prove the correctness of the algorithm, I prove that the algorithm generates the array that contains the least number of inversion.

Proof by Contradiction:
Let A be the array generated by above algorithm
Assume A^{opt} is an array which has the least number of inversion

Assume $A_1 = A_1^{opt}, A_2 = A_2^{opt}, ..., A_{i-1} = A_{i-1}^{opt}, A_i \neq A_i^{opt}$, where $A_i < A_i^{opt}$, as a consequence of the algorithm, and there exists $j > i$, $A_j^{opt} = A_i$ (i, j are not in the constraints).

Let $g = A_j^{opt} = A_i$ (the choice made by my algorithm), $h = A_i^{opt}$, and in interval (i, j), there are x elements $< g$, y elements $> g$, z elements $< h$, r elements $> h$. Because

$g < h$, $x \leq z$ and $y \geq r$

Let $A_i^{opt} = g$ and $A_j^{opt} = h$ (add the greedy choice to A^{opt}), then the number of inversion contributed by g, h, and other elements $\in (i, j)$ is changed as follow:

	increase	decrease
g	x	0
h	0	z+1(the one formed with g)
others $\in (i, j)$	r	y

Overall: $x + r - (z + 1) - y = (x - z) + (r - y) - 1 \leq -1 \implies$ the number of inversion at least decreases by 1.

$\implies A^{opt}$ **does not** have the least number of inversion $\implies$ ***contradiction***

Hence, I have proved that the greedy choice is optimal, which minimizes the number of inversion, and thus minimize the answer of this problem.

To construct A, the algorithm first go through all constraints ($O(N)$), and go through A and M to find unassigned elements ($O(N)$), then run `CountInversion` ($O(N\,lg\,N)$). Thus, the overall complexity is $O(N\,lg\,N)$.

* The pseudo-code is as follow, which applys to both the cases of (4) and (5).

Controllable Ghost Leg

```
input: a set of constraints S, the number of people/prizes N
output: the minimum number of horizontal lines required for the constraints

// The overall complexity is O(n) + On) + O(n) + O(nlgn) = O(nlgn)
function solve(S, N):
    // initializing ...O(n)
    initialize A as an integer array of length N where all elements = −1
    initialize M as an boolean array of length N where all elements is false
    // satisfy the constraints ...O(n)
    for (i, j) in S:
        A[i] ← j
        M[j] ← true

    // handle unassigned prizes and people ...O(n)
    // the value of M[i] indicates whether the i^th prize has been assigned or not.
    p ← 1
    for i from 1 to N:
        // if the i^th person has not been assigned a prize.
        if A[i] = −1:
            // find the minimum available prize
            while p ≤ N and M[p] is true:
                p ← p + 1
            // assign to the person and mark it as assigned
            A[i] ← p
            M[p] ← true

    // call previous function ...O(nlgn)
    return CountInversion(A, 1, N)
```

Problem 7 - Folding Blocks

(1) Let the length of initial block is L, the distance between the block to the left and right boundaries are d_1 and d_2, then the problem is solvable if and only if:

1° $L \mid d_1$

2° $L \mid d_2$

3° $log_2(N \div L) \in (\{0\} \cup \mathbb{N})$

(2) Divided by L, $d_1 + d_2 + 1$ should be power of 2, and $d_1 + d_2 = 2^i - 1$. Thus, if I write $d_1 + d_2$ in binary form, the 0^{th} to $(log_2 N - 1)^{th}$ bits are all 1s. Also, if d_1, d_2 are written in binary form , they should complement each other. That is, for the i^{th} bit, either d_1 or d_2 has 1 at its i^th bit, which is similar to the process of unfold operation: for the i^{th} step, we decide which side should be added 2^i. I take use of bitwise operation in the following implementation.

Folding Block (special case)

```
input: the length of board N,
       the position and length of initial block (pos, len),

output: the sequence of unfold operations that solve the puzzle

function solve(N, pos, len):
    d1 ← pos − 1
    d2 ← N − pos − len + 1;

    N ← N ÷ len
    d1 ← d1 ÷ len
    d2 ← d2 ÷ len

    for i from 0 to (log2 N) − 1:
        if d1 & (1<<i): // if the i^th bit is in d1
            print("unfold to left")
        else: // if the i^th bit is not in d1, then it's definitely in d2
            print("unfold to right")
```

(3) the i^{th} unfolding operation expands either the left or right boundary of the block by $L \times 2^{(i-1)}$ units. Thus, we have two choices every time until one side can no longer be expanded. After the i^{th} operation:

$$d_1' + d_2' = d_1 + d_2 - L \times \sum_{k=0}^{i-1} 2^k \geq 0$$

$$\implies L \times \sum_{k=0}^{i-1} 2^k \leq d_1 + d_2$$

$$\implies 2^i - 1 \leq \frac{d_1 + d_2}{L}$$

$$\implies i \leq log_2(\frac{d_1 + d_2}{L} + 1)$$

On the other hand, since we could get stuck on one side but still be able to unfold to the other side, the number of possiblities $\leq 2^i$ after i^{th} operation. Furthermore, because we can stop at any points, so we should consider every $i, where\ 0 \leq i \leq log_2(\frac{d_1+d_2}{L} + 1)$

Therefore, $\#\ of\ possibilities\ of\ the\ status \leq \sum_{i=0}^{log_2(\frac{d_1+d_2}{L}+1)} 2^i = 2^{log_2(\frac{d_1+d_2}{L}+1)+1} = 2(\frac{d_1+d_2}{L} + 1)$

$0 \leq 2(\frac{d_1+d_2}{L} + 1) \leq 3(d_1 + d_2)\ \forall\ (d_1 + d_2) \geq 1 \implies O(d_1 + d_2)$

(4) Pseudo-code:

Folding Block (general case)

```
input: the length of board N,
       the set S of (pos, len) pairs for every initial block

output: boolean -> whether this puzzle can be solved

// this function simulates the process of unfolding:
// At every state, it has two choices: unfolding to left or right
// then it recursively search all possible sequence of operation, which are all
    possible states, too.
function search(i, len, L, R, left_bound, right_bound, mark):
    // i, len: the current block length is len × 2^i
    // L, R: the current position of the left and right end of the block
    // left_bound, right_bound: boundaries where current block can reach
    if L < left_bound or R > right_bound:  // out of bound
        return

    // if the current block can "connect" one possible status of previous block
    if mark[R + 1] is true:
        mark[L] ← true // mark current left end as a possible status

    // try unfold to left
    search(i+1, len, L − len × 2^i, R, left_bound, right_bound, mark)
    // try unfold to right
    search(i+1, len, L, R + len × 2^i, left_bound, right_bound, mark)

function solve(N, S):
    // initializing ...Θ(N)
    for i from 1 to N:
        mark[i] ← false
    // base case
    mark[N+1] ← true

    // go through all blocks from right to left
    // construct all possible left boundaries
    for i from |S| − 1 to 0:
        if i > 0:
            left_bound ← S[i−1].pos + S[i−1].len
        else:
            left_bound ← 1

        if i < |S| − 1:
            right_bound ← S[i+1].pos − 1
        else
            right_bound ← N

        search(0, S[i].len, S[i].pos, S[i].pos + S[i].len − 1, left_bound,
            right_bound, mark)

    // if the begin of the board can be reached, then there exists a solution
    return mark[1]
```

(5)

- **Overlapping Subproblems:** To determine if a range $[i, N]$ can be solved, I only need to consider the left-most block $B_j = (pos, len)$ where $B_j.pos \geq i$. Because the boundaries of a block's status can never overlap with the other blocks's initial status, only the left-most block have the chance to reach i and the bound of enumeration is set to $[i, S_{j+1}]$. Let the set of all possible status after a series of unfold operation on block B_j , where the left and right ends of block does not exceed L and R respectively be $S(B_j, L, R)$. I recursively define the answer as follow:

  $$Ans(i) = \begin{cases} true & if\ i = N+1\ (base\ case) \\ Ans(R_1+1) \vee Ans(R_2+1) \cdots \vee Ans(R_n+1) & if\ 1 \leq i \leq N \end{cases}$$

  $$...where\ \{R_1, R_2, \ldots R_n\} = \{R \mid (i, R) \in S(B_j, i, B_{j+1}.pos - 1)\}$$

  When I ask for $Ans(R_i+1)$ every time, I again and again enumerate all the possibilities of the status of the block on the right of current block. That forms the overlapping sub-problems. Thus, to avoid recomputing, for every block from right to left, I just enumerate its possible status one time, and if the current status can be "chained" with one of previous status, I record its left boundaries. If the chain can span from 1 to N (i.e. reach 1 from N+1) on board, then the problem is solvable.

- **Optimal Structure:** If we have one of $Ans(R_i + 1)$ is true, then the exists a solution such that $[R_i + 1, N]$ is solvable. As long as the interval is solvable, I can combine one of the solutions with the current status of block.

(6) From the proof in (3), we know that the number of possibilities of the status after some unfold operations are $O(d_1 + d_2)$, and in my algorithm, I enumerate all these possible status by function `search()` for all blocks from left to right. Therefore, the upper bound (Big-O) is:

$$\begin{aligned}
&\sum_{i=0}^{|S|-1} O(d_{1,i} + d_{2,i}) \\
\leq &\sum_{i=0}^{|S|-1} c(d_{1,i} + d_{2,i}) \\
= &2c(\sum_{i=1}^{|S|-2} B_{i+1}.pos - B_i.pos - B_i.len) + c(B_0.pos + N - B_{|S|-1}.pos - B_{|S|-1}.len) \\
\leq &2N = O(N)
\end{aligned}$$

On the other hand, because when initializing `mark` array we must iterate from 1 to N, $T(N) = N + O(N) \geq N = \Omega(N)$, the overall time complexity is $\Theta(N)$.

Discuss with

- B07902064 蔡銘軒
- B07902119 熊育霆
- B07902132 陳威翰