Collaboration policy

I've discussed with B12902098, and also inspired by other B12.

Problem 5. Nathan at Small Circle Village

Nathan loves bubble tea

(a) (3 points)

Solution.
This is a classic 0/1 knapsack problem.
We can define $dp[i][j]$ to be the maximum happiness Nathan can obtain when he had j dollars with him and only considered the first i bubble tea.
Hence, we have the base case:

$$dp[0][j] = 0, \ \forall \ 0 \leq j \leq M$$

To solve all the dp values for $0 < i \leq N$:

$$dp[i][j] = \begin{cases} \max\{dp[i][j-1], dp[i-1][j], dp[i][j-c_i] + h_i\}, & j \geq c_i \\ \max\{dp[i][j-1], dp[i-1][j]\}, & otherwise \end{cases}$$

The answer will be $dp[N][M]$.
The time complexity will be $O(1) \times O(NM) = O(NM)$.

□

Nathan and his good days

(b) (2 points)

Solution.
Notice that we always want to buy K cups of bubble tea, since the happiness is always positive.
Also, we always want to buy the bubble tea with higher happiness as we can.
Hence, we can design a greedy algorithm to solve the problem.
First, we sort N cups of bubble tea by h_i in $O(NlogN)$.
Second, we choose the top K cups of bubble tea with higher happiness Nathan can obtain in $O(K)$.
Finally, the answer will be the sum of h_i for the top K cups of bubble tea.
The time complexity will be $O(NlogN) + O(K) = O(NlogN) \in o(N^2)$.

□

Nathan feeling EMO

(c) (6 points in total)

Solution.
This will be a solution to (c - 2).
Define $dp[i][j]$ to be the maximum happiness Nathan can obtain if he buys i cups of bubble

tea while he buys the last one at shop j.
First, we set $dp[0][0] = 0$.
To solve all the dp values $\forall\ 0 < i \leq N,\ i \leq j \leq i \times L$:

$$dp[i][j] = \max_{\min\{j-L, i-1\} \leq k < j} \{dp[i-1][k]\} + h[j]$$

We can maintain the max term with monotonic queue, which can be implemented using deque.
Each element in the deque will be an index k, which is used to maintain $dp[i-1][k]$ and due time $k + L + 1$ (After that the element should be deleted).
We will always try to keep the available maximum value at the front of the deque.

Algorithm 1: Monotonic queue

$dp[0][0] \leftarrow 0$;
for $i = 1$ *to* K **do**
    deque $\leftarrow$ [ ];
    for $j = i - 1$ *to* $\max\{i \times L, N\}$ **do**
        while *deque is not empty* **do**
            $k \leftarrow$ the front element of deque;
            if $k < j - L$ **then**
                Pop the element from the front of deque;
            end
            else break;
        end
        if $j \geq i$ **then**
            $x \leftarrow$ the front element of deque;
            $dp[i][j] \leftarrow dp[i-1][x] + h[j]$;
        end
        while *deque is not empty* **do**
            $k \leftarrow$ the back element of deque;
            if $dp[i-1][j] \geq dp[i-1][k]$ **then**
                Pop the element from the back of deque;
            end
            else break;
        end
        Push j into the back of the deque;
    end
end

The algorithm have the time complexity of $O(NK)$ since every element will be push into (pop out from) deque at most once respectively.
The answer will be $\max_{N-L<j\leq N} dp[K][j]$ which can be calculated in $O(L)$.
Hence, the complexity will be $O(NK) + O(L) = O(NK)$.

□

Nathan and Fysty

(d) (4 points)

Solution.
Consider using dynamic programming.
Define $dp[i][j][0/1/2]$ as follows: $dp[i][j]$ means maximum happiness Nathan can obtain if only consider $h_1, ..., h_i$, and we have chosen h_i, while we have bought j of them.
The last argument makes a slight difference as follows:

- $dp[i][j][0]$ means we haven't done anything about the swap.
- $dp[i][j][1]$ means we have chosen one of the j elements h_k, where $k \le i$, to be swapped afterward, so we regard h_k as 0 in this case.
- $dp[i][j][2]$ means we have swapped two elements already.

Base case: $dp[0][0][0] = 0$.
In particular, to simplify the calculation, we will make all the undefined values ($j > i$ or the last argument is greater than j, etc.) to be ∞, so it will never become a part of the answer.
To calculate all the dp values recursively:

- $dp[i][j][0] = \max\limits_{i-L \le k < i} dp[k][j-1][0] + h_i$
- $dp[i][j][1] = \max\limits_{i-L \le k < i} (\max\{dp[k][j-1][1] + h_i, dp[k][j-1][0]\})$
- $dp[i][j][2] = h[i] + \max(\max\limits_{i-L \le k < i} dp[k][j-1][2], \max\limits_{i-L \le k < i} \{dp[k][j-1][1] + \max\limits_{k < k' < i} h[k']\})$

for all $1 \le i \le N$ and $1 \le j \le \min\{i, K\}$.
Since we only consider the case that swaps the element front, we can **reverse** the array and apply the algorithm again to obtain dp'.
The answer will be:

$$\begin{aligned}\max\{ &\max_{N-L < i \le N} \{\max(dp[i][K][0], dp[i][K][2], dp'[i][K][0], dp'[i][K][2])\}, \\ &\max(dp[N][K][1], dp'[N][K][1]) + h[N]\}\end{aligned}$$

The time complexity will be:

$$O(NK) \times O(L) = O(NKL)$$

□

(e) (8 points in total)

Solution.

This will be a solution to (e - 2).

By observation, we can easily know that we only have one way to obtain H by choosing the greatest K element to obtain H because each element in $\{h_i\}$ is distinct.
In other words, we **need** to get the greatest K element if we want to obtain H.
Hence, the problem can be seen as follows:
If we denote the greatest K elements to be **Good Elements**, we have Good Elements and their position x_i from small to large, while we need to ensure that $x_{i+1} - x_i \le L$.
First, we sort $\{h_i\}$ to obtain the Good Elements and their positions in $O(N \log N)$.
Define $dp[i][j]$ as the minimum number of swaps if we choose the i^{th} element and the segment $[1, i]$ is legal, while j means we can provide at least j Good Elements for later swaps ($-T \le j \le T$, negative means we need at most $|j|$ Good Elements).
Consider $cnt_{l,r}$ to be the number of Good Elements in $[l, r]$.
And we define

$$g(x, y) = \begin{cases} \min(|x|, |y|), & \text{if } xy \le 0 \\ 0, & otherwise \end{cases}$$

To calculate how many swaps we need to transit from $j = x$ to $j = y$.
In particular, to simplify the calculation, we will make all the undefined values ($j > cnt_{1,i}$ or j is not in the range $[-T, T]$, etc.) to be ∞, so it will never become a part of the answer.
Base case:

$$dp[0][j] = \begin{cases} 0, & \text{if } j \le 0 \\ \infty, & otherwise \end{cases}$$

To calculate dp values recursively:

$$dp[i][j] = \begin{cases} \min(dp[i+1][j], \min\limits_{i-L \le k < i} \{dp[k][j - cnt_{k,i-1}] + g(j - cnt_{k,i-1}, j)\}), i \text{ is a Good Element} \\ \min(dp[i+1][j], \min\limits_{i-L \le k < i} \{dp[k][j - cnt_{k,i-1} - 1] + g(j - cnt_{k,i-1} - 1, j)\}), \text{otherwise} \end{cases}$$

In particular, cnt is easy to compute if we iterate k from large to small.
Finally, to obtain the answer, we evaluate the inequality $\min\limits_{N-L < i \le N} \{dp[i][0]\} \le T$, if it's true then it's possible for Nathan to obtain the happiness H, otherwise it's impossible.
Hence, the time complexity will be $O(NK) \times O(L) + O(N \log N) = O(NLK + N \log N)$.

□

Problem 6. ADA FTP server

Download homework quickly

(a) (3 points)

Solution.
Notice that we can compute the downloading time m_i by:

$$m_i = \frac{s_i}{\min\{u, d_i\}}$$

It's easy to compute that

$$m_1 = 12$$

$$m_2 = 5$$

(1) The ADA FTP server first serves student 1

$$c_1 = m_1 = 12$$

$$c_2 = c_1 + m_2 = 17$$

Thus, the average would be

$$c_{avg} = \frac{c_1 + c_2}{2} = 14.5$$

(2) The ADA FTP server first serves student 2

$$c_2 = s_2 = 5$$

$$c_1 = c_2 + m_1 = 17$$

Thus, the average would be

$$c_{avg} = \frac{c_1 + c_2}{2} = 11$$

□

(b) (4 points)

Solution.
It's trivial that if we minimize $\sum_{i=1}^{n} c_i$, we also minimize c_{avg} at the same time. So the later one is the one we do.
First, compute the time m_i in $O(n)$.
And here's a simple observation,

$$c_i = m_i + \text{sum of } m_j, \text{ for all } j \text{ is finished before } i$$

Thus, if a task i is finished before j, c_j will increase by m_i.
When we have a specific order $\{ord_i\}$, which is a permutation from 1 to n, we can represent $\sum_{i=1}^{n} c_i$ as follows:

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} (n - ord_i + 1) \times m_i$$

Hence, we can design a greedy algorithm that finishes task with less t_i.
To prove its correctness, assume that we already have $\{ord_i\}$ ordered by m_i, where $\forall\ 1 \leq ord_i < ord_j \leq n :\ m_i < m_j$.
If we swap ord_a, ord_b, where $1 \leq ord_a < ord_b \leq n$, the term $\sum_{i=1}^{n} c_i$ will increase by $(ord_b - ord_a)(m_b - m_a) > 0$.
To implement the algorithm, we can sort $\{m_i\}$ and order them by the value m_i in $O(n \log n)$.
Hence, the complexity will be $O(n) + O(n \log n) = O(n \log n)$.

□

(c) (4 points)

Solution.
We should never suspend the current file transmission.
To prove its correctness, knowing that we should always finish the task with smaller m_i first from (b), if we suspend the current task at any time stamp, we should still resume the same task (since it still has the smallest m_i).
Hence, we can solve the problem using exactly the same method in (b) within $O(n \log n)$.

□

(d)

Solution.
Notice that t_i means that i can only be taken into consideration after t_i.
Different from (c), this time we do have reasons to suspend a task since the greatest i can be added after specific time stamp t_i.
We can design a new greedy algorithm as follows:
First, we sort i by t_i in $O(n \log n)$.
Second, iterate i and wisely choose which task to be done first (Reconsider the greatest option when we add a new task i at time stamp t_i).
To make it clear, we will maintain a priority queue (Implemented by a min heap) which gives us the smallest m_i. The algorithm works as follows:
We can prove the correctness since we always try to finish the task with smallest m_i.

Algorithm 2: Greedy

```
pq ← an empty min heap ;
left ← n;
cur ← 0;
sum ← 0;
for i = 1 to n do
    while pq is not empty do
        k ← the element at the top of pq;
        pop out the element at the top of pq;
        if k + cur ≤ t_i then
            /* Done the jobs with smallest m_i as many as possible */
            sum ← sum + k × left;
            left ← left − 1;
            cur ← cur + k;
        end
        else
            k ← k − (t_i − cur);
            push k back into pq ;                  // suspend the task
            break;
        end
    end
    push m_i into pq ;                             // take m_i into consideration
end
while pq is not empty do
    k ← the element at the top of pq;
    pop out the element at the top of pq;
    sum ← sum + k × left;
    left ← left − 1;
end
```

To evaluate complexity:

1. We only have to do $O(n)$ pushes/pops to the priority queue since we will at most put two elements in priority queue in a single iteration in i, and we have to pop all the elements out.
   Thus, The complexity of Algorithm 2 will be $O(n) \times O(\log n) = O(n \log n)$ (By using a heap).

2. Recall that we first sort the array in $O(n \log n)$.

Hence, the time complexity to solve problem (d):

$$O(n \log n) + O(n \log n) = O(n \log n)$$

□

Arrange the ADA FTP server

(e)

Solution.
We have to get some u_i, that every u_i is not adjacent.
Hence, can design a dynamic programming algorithm as follows.
Define $dp[i]$ as the maximum sum if we only consider $u_1, ..., u_i$.
Base case: $dp[0] = 0, dp[1] = u_1$.
To calculate all dp values for $1 < i \leq n$:

$$dp[i] = \max\{dp[i-1], dp[i-2] + u_i\}$$

If we choose i as a part of the answer, it will be optimal if we contain the optimal solution that only contains $u_1, ...u_{i-2}$. Otherwise, the answer will be the optimal solution which only contains $u_1, ..., u_{i-1}$.
Finally, the answer will be $dp[n]$.
And the complexity of the algorithm will be $O(n)$.

□

(f)

Solution.
Define $dp[i]$ to be the maximum sum we can obtain if we only consider $u_1, ...u_i$.
Define $f(i)$ to be the greatest j such that $x_i - r_i > x_j$ (Since r_i is increasing, $\max(r_i, r_{f(i)})$ always equals to r_i), which means the last valid j that i and j don't interfere each other. If there doesn't exist a valid j, we define $f(i) = 0$.
Since $dp[i]$ is increasing (Considering more and more elements), $f(i)$ will always be optimal to transit to $dp[i]$.
Notice that we can use binary search to get $f(i)$ in $O(\log n)$.
Base case: $dp[0] = 0$.

To calculate all dp values for $0 < i \leq n$:

$$dp[i] = \max\{dp[f(i)] + u_i, dp[i-1]\}$$

If we choose i as a part of the answer, it will be optimal if we contain the optimal solution that only contains $u_1, ...u_{f(i)}$. Otherwise, the answer will be the optimal solution that only contains $u_1, ..., u_{i-1}$. Finally, the answer will be $dp[n]$.
And the complexity of the algorithm will be $O(n) \times O(\log n) = O(n \log n)$.

$\square$