CS240 Homework #2

Zhiqiang Xie 77892769

Collecting Toys

1. $p_{i,j} = \frac{j}{n}p_{i-1,j} + \frac{n+1-j}{n}p_{i-1,j-1}$, for $i \geq 2$ and $1 \leq j \leq n$
2. As for $i = 1, j = 0$, we have the base cases: $p_{1,0} = 0, p_{1,1} = 1$ and $p_{i,j} = 0$ for $j > i$

   Therefore, any $p_{i,j}$ can be converted to the combination of the base cases above and then we can calculate it out.

Knapsack II

Since $n > v_i > 0$, we get: $\sum_{i=1}^{n} v_i < n^2$. Besides, n, v_i are integers, but w_i, W are real number.

We can and have to apply DP by constructing table in objects and values instead of weights.

Python

```
values = [0, v1, v2, ..., vn]
weights = [0, w1, w2, ..., wn]
table = [[float("Inf")]*(n**2) for i in range(n+1)]
max_value = 0
for i in range(1, n):
    for v in range(n**2):
        if v > values[i]:
            table[i,v] = min(table[i-1,v], weights[i]+table[i-1,v-values[i]])
        else:
            table[i,v] = min(table[i-1,v], weights[i])

        if table[i,v] < W: # W is the capacity of knapsack
            max_value = max(max_value, v)
```

The variable `max_value` tells us the maximum value we can achieve with the capacity constrain.

The time complexity $T(n) = \Theta(1) + n * n^2 * \Theta(1) \in \Theta(n^3)$

Counting Friends

Firstly, let's sort the scores of all students by x_i in **non-decreasing** order.

- This step costs $O(n \log n)$ time.

Now, the only slight difference between this problem and the *counting inversion* problem is that the indices of elements always increase, but our sorted $x_1, \ldots, x_i, \ldots x_n$ isn't.

Now I'll present how to solve this problem using an AVL tree with handling duplicates.

- Create an empty AVL tree and initialize the pairs of friends to be 0.
- Every node in the AVL tree records the number of nodes in its right subtree, and this attribute will be updated in every insertion and balancing steps.
    - Here we may have better attribute candidates such as the size of the tree rooted with this node, which can be updated more easily in balancing step. And it's also convenient to calculate the number of nodes in its right subtree.
- We use y_i as the value of nodes and iterate all i to insert them into the AVL tree.
- Every time we do insertion into AVL tree, we traverse the tree from the root to a leaf by comparing every node in the path with y_i. When y_i is smaller than the current node, we increase the pair count by 1 plus the number of nodes in the right subtree of the current node. When y_i is larger than the current node, keep traversing and update the attribute of the current node.
    - If the node we are going to insert satisfies $x_i = x_{i-1}$, we should deduct 1 from the count when we increase it.
    - If the node we are going to insert has the same value as the current node, just move along the path to right subtree. Remember to update the attribute too.

Therefore, we'll get the correct number of pairs of friends.

The time complexity $T(n) = O(n \log n) + n * O(\log n) \in O(n \log n)$

XOR Convolution

Since all integer can be writed in binary form with the length (digits) of $\lfloor \log_2 n \rfloor + 1$, such as $8 = 1000$.

As for $c_i = \sum_{j \oplus k = i} a_j b_k$,

$0 = xxxx \oplus xxxx, 1 = xxx1 \oplus xxx0, 2 = xx1x \oplus xx0x, 3 = xx11 \oplus xx00 = xx10 \oplus xx01 \ldots$

This series of expressions is actually general (commutation is regarded as the same) and you can find that: $i \geq |j - k|$. Since we need the corresponding distinct bits to get the target, the maximum of difference could be i.

Therefore, we can set $C = C_0, C_1$, where $C_0 = c_0, c_1, \ldots, c_{\frac{m}{2}-1}, C_1 = c_{\frac{m}{2}}, \ldots, c_{m-1}$.

- Then, $A = A_0, A_1$, $B = B_0, B_1$.

Therefore, $C_0 = A_0 * B_0 + A_1 * B_1, C_1 = A_0 * B_1 + A_1 * B_0$, since $(i \geq |j - k|)$

Here the time complexity is $T(n) = 4 * T(n/2) + O(n) \in O(n^2)$.

However, here are two more expressions inside here:

- $C_0 = ((A_0 + A_1)(B_0 + B_1) + (A_0 - A_1)(B_0 - B_1))/2$

- $C_1 = ((A_0 + A_1)(B_0 + B_1) - (A_0 - A_1)(B_0 - B_1))/2$

Thus, the time complexity $T(n) = 2T(n/2) + O(n) \in O(n \log n)$

DNA Pattern Recognition

Our goal is actually to find the maximum matching between the two strings, and then we can get the minimum number of changes.

The most convenient way to find the maximum matching in signal processing is to calculate the convolution of two signals. Here we hold the similar method.

Firstly, we need to quantify the four different signal pulse: A, T, G, C.

- $S_A[i] = 1, \forall S[i] = A$, else $S_A[i] = 0$, for all $0 \leq i \leq n-1$
- $S_T[i] = 1, \forall S[i] = T$, else $S_T[i] = 0$, for all $0 \leq i \leq n-1$
- $S_G[i] = 1, \forall S[i] = G$, else $S_G[i] = 0$, for all $0 \leq i \leq n-1$
- $S_C[i] = 1, \forall S[i] = C$, else $S_C[i] = 0$, for all $0 \leq i \leq n-1$
- $P_A[i] = 1, \forall P[i] = A$, else $P_A[i] = 0$, for all $0 \leq i \leq n-1$
- $P_T[i] = 1, \forall P[i] = T$, else $P_T[i] = 0$, for all $0 \leq i \leq n-1$
- $P_G[i] = 1, \forall P[i] = G$, else $P_G[i] = 0$, for all $0 \leq i \leq n-1$
- $P_C[i] = 1, \forall P[i] = C$, else $P_C[i] = 0$, for all $0 \leq i \leq n-1$

Then we need to convolve these sub-signal to find the maximum matching.

$M_{max} = max(conv(S_A, P_A) + conv(S_T, P_T) + conv(S_G, P_G) + conv(S_C, P_C))$

However, it costes $T(n) = \Theta(m) + \Theta(n) + 4 * \Theta(m * n) + \Theta(n) \in O(n^2)$

Here's one way to accelerate the convolution step: the convolution in the time domain is equivlent to a multiplication in the frequency domain.

$M_A = IFFT(FFT(S_A) * FFT(P_A))$

$M_T = IFFT(FFT(S_T) * FFT(P_T))$

$M_G = IFFT(FFT(S_G) * FFT(P_G))$

$M_C = IFFT(FFT(S_C) * FFT(P_C))$

$M_{max} = max(M_A + M_T + M_G + M_C)$

The time complexity now is
$T(n) = \Theta(m) + \Theta(n) + 4 * (O(n \log n) + O(m \log m)) + 4 * (m + n) \log(m + n) + \Theta(n)$

Since $\sqrt{n} < m < n - \sqrt{n}, T(n) \in O(n \log n)$

2D Inversions

In this problem, I'll reuse my AVL tree presented in Counting Friends problem.

1. The only difference between this subproblem with the former one is this constraint:
   - $y_i \geq y^{'} > y_j$, where $y^{'}$ is a fixed constant.

   We could just modify the attribute to be the number of nodes whose $y_i \geq y^{'}$ in its right subtree.

   Thus, the counting of half-inversions is increased by the number recorded with double constrains.

   The time complexity remains the same: $T(n) = O(n \log n)$

2. This problem is almost the same as the one above.
   - Firstly, we sort the pairs by x_i in non-increasing order.
   - Then we use y_i to be the value of TreeNode, set indices i, j to be the constraints.
     - Be careful about the duplicates, and we have discussed in Counting Friends problem.

   The time complexity remains the same $T(n) = O(n \log n)$

3. This one is quite different, one more dimension of data are supposed to take into comparing.

   As a baseline, I'll compare some different method to solve this problem.

   AVL + local array
   - This is the most intuitive one from our previous algorithm. We store the number of nodes in its right subtree as well all value of y of them.
   - Every encounter we search all y in local array and determine the number to be added to counting.
   - The time complexity is $T(n) = O(n^2)$

   AVL + local tree
   - Optimize for local search.
   - The time complexity of average case is $T(n) = O(n \log^2 n)$
   - It's easy to fall into worst case though.

   AVL + local AVL
   - Meaningless, it's inefficient to maintain two AVL tree since the local one has to balance itself too often.
   - And it's troublesome to update the attributes.
   - The time complexity might be $T(n) = O(n \log^2 n)$ though it may not work at all.

   MergeSort + local AVL
   - Apply the divide and conquer method used in counting inversion here, and then maintain a local AVL tree in the merge stage to determine how to increase the counting.
   - This method is concise and efficient, the time complexity is $T(n) = O(n \log^2 n)$

   2-D Binary Index Tree
   - Here is no time for it, I'll update it later.