leetcode.com /discuss/general-discussion/1129459/patience-sorting-dp-optimisation

Patience Sorting (DP Optimisation)

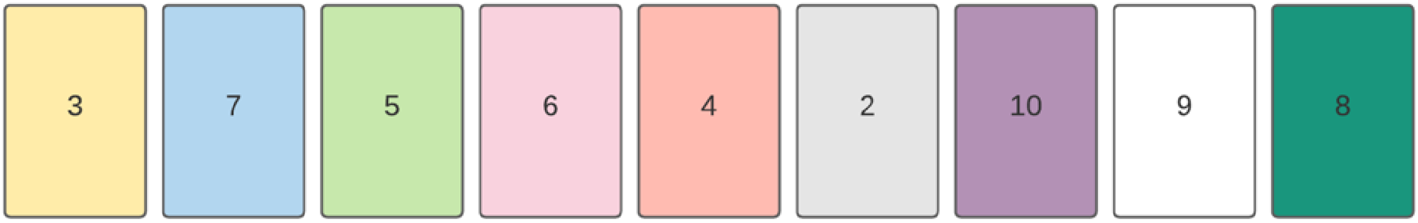
8-10 minutes

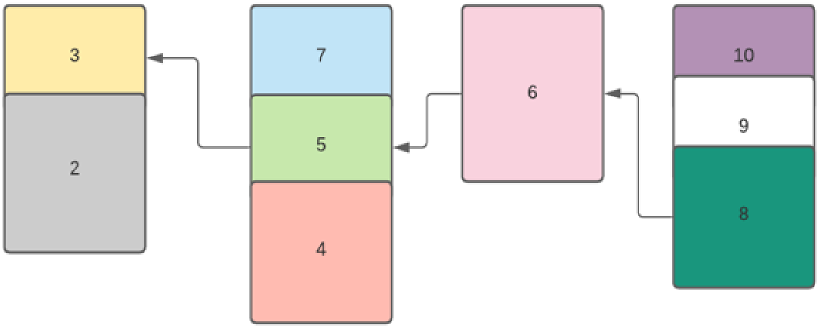
Patience Sorting is a powerful technique that can transfrom your solutions to **Longest Increasing Subsequence** type **Dynamic Programming problems** from `O(n^2)` to`O(nlogn)` complexity. It will allow you to solve hard DP problems.

The agenda of this post

- Introduce you to the Patience Sorting algorithm through a card game.
- Learn how to apply it to DP problems.
- Show the application of this algorithm through a number of examples.

Game of Solitaire





The goal of this game is to form as few piles as possible following the below rules,

1. You cannot place a higher value card on top of a lower value card,
2. You will place the card on the leftmost pile that fits.
3. If you cannot find a pile that can accomodate your card, then you can start a new pile.

In the above example diagram we have cards valued [3, 7, 5, 6, 4, 2, 10, 9, 8],
Steps

- The first card `3` begins a new pile, lets call it `Pile-1`.
- The next card `7` is greater than the topmost card of the `Pile-1`, so `7` begins a new pile. Lets call it `Pile-2`.
- The next card `5`, can be placed on top of the `Pile-2` as `5<7`.
- The next card `6`, cannot be placed on top any of the previous piles, so it begins a new pile `Pile-3`.
- The next card `4`, can be placed in top of `Pile-2` as `4<5`.
- The next card `2` can be placed on top of `Pile-1` as `2<3`. (*Note that `2` could have been placed on top of `Pile-1`, `Pile-2` or `Pile-3`, but we should choose the left most pile that fits. Which is `Pile-1` in this case*)
- The next card `10` forms a new pile `Pile-4`.
- The next card `9` can be placed on top of `Pile-4`.
- The next card `8` can be placed on top of `Pile-4`.

The total number of piles gives the length of the longest increasing subsequence.

In this case, the length of the longest increasing subsequence is 4.

You can recover the longest increasing subsequence if you maintain back pointers

In this case, card `8` has a pointer to the top of previous pile, card `6`. Card `6` has a pointer to the top of previous pile, card `5` (The top of `Pile-2` when card `6` was inserted). Card `5` has a poinnter to the top of previous pile, card `3` (The top of `Pile-1` when card `5`was inserted)

So `[3, 5, 6, 8]` is one of the longest increasing subsequences.

How to Implement

This is a greedy algorithm and uses binary search. You will use binary search to find the left-most pile that can accomodate a card.

```
class Solution {
public:
```

```
    int lengthOfLIS(vector<int>& nums) {
        int n = nums.size();
                // seq stores the piles
        vector<int> seq;

                // create the first pile
        seq.push_back(nums[0]);

                // Go through each card
        for(int j=1; j<n; j++){

                        // Find the left-most pile that can accomodate this
card
            int idx = binSearch(seq, nums[j]);

            if(idx == -1){
                                // If no such pile exists, then create a new
pile
                seq.push_back(nums[j]);
            }else{
                                // If a pile is found that can accomodate this
card,
                                // then place this card on top of that pile
                seq[idx] = nums[j];
            }
        }

                // The number of piles is the length of the longest increasing
subsequence
        return seq.size();
    }

        // Binary search to find the left-most pile that can accomodate a card
        // If a pile is found, then it returns the index to that pile
        // If a pile is not found, then return -1
    int binSearch(vector<int> &seq, int i){
```

```
        int l=0, h = seq.size()-1, m;
        int res = -1;

        while(l<=h){
            m = l+(h-l)/2;
            if(seq[m] >= i){
                res = m;
                h = m-1;
            }else{
                l = m+1;
            }
        }

        return res;
    }
};
```

LIS steps

- Create a container to store the piles.
- Create the first pile.
- Go through each card.
- Find the left-most pile that can accomodate this card.
- If no such pile exists, then create a new pile.
- If a pile is found that can accomodate this card, then place this card on top of that pile.
- The number of piles is the length of the longest increasing subsequence.

Binary Search

- Binary search to find the left-most pile that can accomodate a card.
- If a pile is found, then it returns the index to that pile.
- If a pile is not found, then return -1.

1671. Minimum Number of Removals to Make Mountain Array

https://leetcode.com/problems/minimum-number-of-removals-to-make-mountain-array/

Hard Problem

```
class Solution {
public:
    int minimumMountainRemovals(vector<int>& nums) {
        int n = nums.size();
                // Keeps track of the LIS for each element in nums
        vector<int> is(n);
                // Stores the piles
        vector<int> cont;
                // Keeps track of the Longest Decreasing Subsequence from each
element in nums
        vector<int> ds(n);


        // Calculate the increasing sequence
                // Create the first pile
        cont.push_back(nums[0]);
                // The length of LIS for first element is 1
        is[0] = 1;

                // Go through each card
        for(int i=1; i<n; i++){
                        // Find the leftmost pile that can accomodate this card
            auto it = lower_bound(cont.begin(), cont.end(), nums[i]);
            if(it == cont.end()){
                                // If such a pile does not exist
                                // then create a new pile
                cont.push_back(nums[i]);
                is[i] = cont.size()-1;
            }else{
                            // If such a pile exists
                                // Then put is card on top of that pile
                *it = nums[i];
                is[i] = distance(cont.begin(), it);
            }
        }
```

```
        // Calculate the decreasing sequence
                // The same steps as above but for the decreasing sequence
        cont.clear();
        cont.push_back(nums[n-1]);
        ds[n-1] = 1;
        for(int i=n-2; i>=0; i--){
            auto it = lower_bound(cont.begin(), cont.end(), nums[i]);
            if(it == cont.end()){
                cont.push_back(nums[i]);
                ds[i] = cont.size()-1;
            }else{
                *it = nums[i];
                ds[i] = distance(cont.begin(), it);
            }
        }

                // Find the longest mountain array
        int ans = INT_MAX;
        for(int i=1; i<n-1; i++){
            if(is[i] && ds[i])
                ans = min(ans, n-(is[i]+ds[i]+1));
        }

        return ans;
    }
};
```

Read though the comments in the code for understanding the implementation details

Russian Doll Envelopes
https://leetcode.com/problems/russian-doll-envelopes/
Hard Problem

```
class Solution {
public:
    int maxEnvelopes(vector<vector<int>>& envelopes) {
```

```
                // container to store the piles
        vector<int> seq;

                // sort the envelopes
                // If two envelopes have the same width, then the envelope with
the largest height is placed before
                // This is because we will apply the patience sort algorithm of
the heights
        sort(envelopes.begin(), envelopes.end(), [](vector<int> a, vector<int>
b){
            return ((a[0] < b[0]) || (a[0] == b[0] && a[1] > b[1]));
        });

                // For each envelope
        for(vector<int> e: envelopes){
                    // Find the left most pile that can accomodate the envelope
            int idx = binSearch(seq, e[1]);
            if(idx == -1){
                                // If no such pile is found, then create a new
pile
                seq.push_back(e[1]);
            }else{
                                // If such a pile is found then
                                // Make this envelope the top of the pile
                seq[idx] = e[1];
            }
        }

                // The number of piles is the length of the LIS
        return seq.size();
    }

        // Binary search to find the left most pile that can accomodate a
envelope
        // If such a pile exists, then return the index of the pile
        // Else return -1
```

```
    int binSearch(vector<int> &seq, int b){
        int low = 0, high = seq.size()-1, mid, res = -1;

        while(low <= high){
            mid = low + (high-low)/2;
            if(seq[mid] >= b){
                res = mid;
                high = mid-1;
            }else{
                low = mid+1;
            }
        }

        return res;
    }
};
```

Read though the comments in the code for understanding the implementation details

*If you liked the post, please don't forget to **upvote**.*