1. Exercise 13 Page 194

Algorithm

We compute $w_i \cdot t_1 \cdot t_2 \cdots t_{i-1} \cdot t_{i+1} \cdots t_n$ for every customer i. Then order them in descending order. Finally, picking corresponding job i in this order minimize the weighted sum of the completion time. We can first multiply every weight w_i by $t_1 \cdots t_n$ and divide it by t_i, which takes $O(N)$. Then we can sort these values in $O(NlogN)$. Thus, the time complexity of this algorithm is $O(NlogN)$.

Proof (using Exchange Arguments)

We will prove the algorithm above is optimal. Suppose there exists a different order of jobs whose completion time is shorter. Then it must contain a pair of jobs i and j where job i is completed before job j in our algorithm while job j is completed before job i in proposed algorithm. Let job i and j be an adjacent pair. Then the weighted sum of completion time from job i to job j in our algorithm is $w_i t_i + w_j(t_i + t_j)$. The weighted sum of completion time from job j to job i in proposed algorithm is $w_j t_j + w_i(t_j + t_i)$. From the nature of our algorithm, $w_i \cdot t_1 \cdot t_2 \cdots t_{i-1} \cdot t_{i+1} \cdots t_n < w_j \cdot t_1 \cdot t_2 \cdots t_{j-1} \cdot t_{j+1} \cdots t_j$ (i.e. $w_i t_j \geq w_j t_i$). Note $(w_i t_i + w_j(t_i + t_j)) - (w_j t_j + w_i(t_j + t_i)) = w_j t_i - w_i t_j \leq 0$. In other words, we can repeatedly swapping such jobs i and j without increasing but even decreasing the weighted sum of completion time, and transform the proposed solution into our solution.

2. Exercise 17 Page 197

Algorithm

Iterate over each job. During each iteration, remove all the jobs that have overlaps with selected interval I_i. Next, we cut the time-line from any time within I_i. Then we perform Interval Scheduling Algorithm on this new time-line (i.e. picking an interval that ends first) to obtain an optimal solution including I_i. After iterating over all the jobs, we have n optimal solutions with each including I_i. Finally, we pick an optimal solution that completes the most number of jobs. We iterate over every job and during each iteration, we potentially need to process $n-1$ jobs at most for applying Interval Scheduling Algorithm. Thus, the time complexity of this algorithm is $O(N^2)$.

Proof

We will prove that our algorithm produces the optimal solution by contradiction. Suppose there is another solution that completes more jobs. Then this proposed solution must contain an interval I_x. Since our algorithm is choose the optimal solution out of every optimal solution containing I_i, our solution will achieve at least what I_x will achieve. This contradicts our assumption that I_x completes more jobs than our solution.

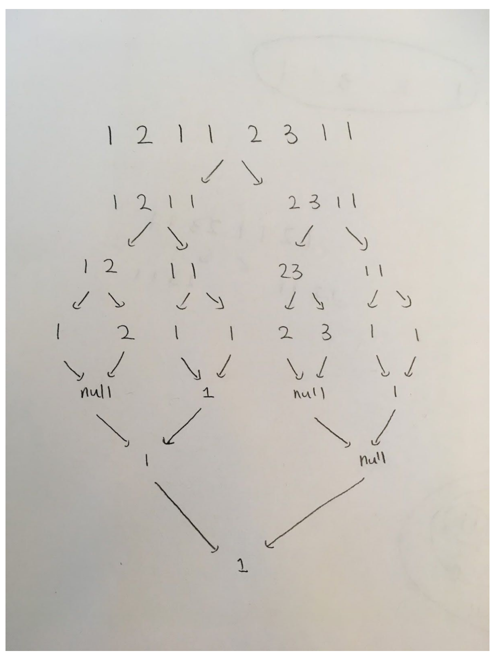
3. Exercise 3 Page 246

Idea

We want to find a card who is equivalent (having the same class) to more than $n/2$ of the cards. Notice if we divide the cards into half (Let's call them left pile and right pile). Either more than half of left pile or more than half of the right pile must contain this class.

Algorithm

We divide the cards into half (Let's call them left pile and right pile). We recursively divide the cards into half for the left pile. If there is only one card, return that card. If there are two cards, check if they are equivalent. If they are, arbitrary pick one of the cards, and return it. If a card is returned from the recursive call, then check if this card belongs to majority class by testing against left pile and right pile. It no card is returned from the recursive call, we recursively divide the cards into half for the right pile. If a card is returned from the recursive call, then check if this card belongs to majority class by testing against left pile and right pile. Return a card with majority class if found.



Let $T(n)$ be the time complexity of this algorithm. Then

$$T(n) = 2T(n/2) + 2n = 2^2T(n/2^2) + 2 \cdot 2n = 2^3T(n/2^3) + 3 \cdot 2n = \cdots = 2^{logn}T(1) + logn \cdot 2n = n + 2nlogn$$

Thus, the time complexity of this algorithm is $O(NlogN)$.

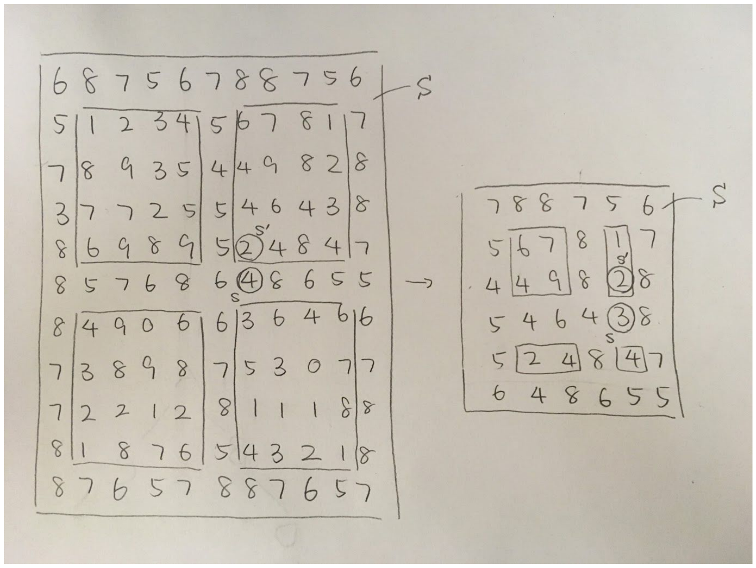
Proof

The correctness of this algorithm is already discussed in the idea section above. If there exists a majority class, then at least one of left half pile or right half pile must contain this majority class. By the nature of our algorithm, it will find this majority class.

<u>4. Exercise 7 Page 248</u>

Algorithm

Examine boundary, middle row, and middle column elements of the grid (Let's call this region S). Note by doing so, we have four quadrant regions.



Pick the smallest element s in S. If that element is local minimum, then return it. Otherwise, there must be at least one element s' adjacent to s and smaller than the s. We claim that a local minimum must exist within the quadrant that s' belongs to.

Proof of Claim

Suppose s' is an element within the quadrant, adjacent to the boundary, and smaller than any element in the boundary. Suppose on the contrary, there isn't a local minimum in the quadrant (i.e. every element in the quadrant is greater than its adjacent elements). Then, every element adjacent to s' is greater than s'. Since s' is smaller than any element in the boundary, s' is the local minimum. This contradicts our assumption that there is no local minimum in the quadrant.

Algorithm (Continued)

We recursively examine boundary, middle row, and middle column of selected quadrant including the boundary. During each iteration, we need to find the minimum of the boundary, middle row, and middle column, which takes less than $6n$ comparisons. Every time we recurse, we reduce the problem size by half. Thus, suppose the time complexity of this algorithm is $T(n)$. Then, $T(n) = T(n/2) + 6n = 6n + 3n + \cdots + 1 = 6n(1 - (1/2)^{logn})/(1 - 1/2) = 12n(1 - (1/n)) \simeq 12n$. Thus, the time complexity of this algorithm is $O(N)$ as desired.

5.

Algorithm

We want to find a pivot point of the array (i.e. the smallest element in the array). Then the index of that element is K. The algorithm is the following. Pick left most ($left$), right most ($right$), and middle element (mid) of the array.

If $left \leq mid \leq right$:

  This indicates the array is sorted, and thus the pivot is the first element.

Else If mid is local minimum:

  Then mid is the pivot.

Else:

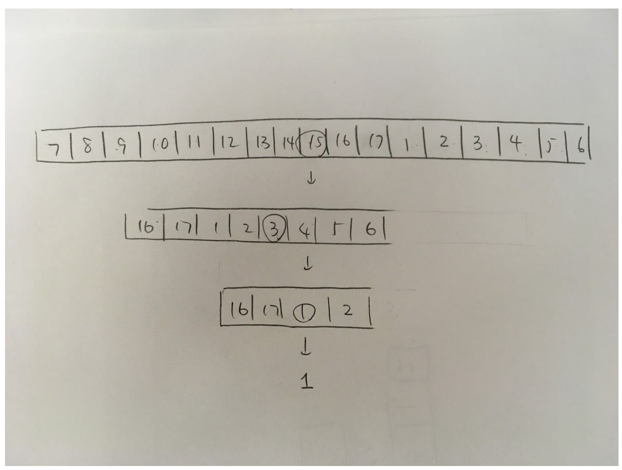
  If $left \geq mid$:

    Then we claim that the pivot must exist in the left half of the array.

  Else:

    It must be the case where $mid \geq right$, and we claim the pivot must exist in the right half of the array.

  Recursively perform this algorithm for selected half of the array



This algorithm divides the problem size into half every iteration, so the time complexity is $O(logN)$.

Proof of Claim

When we divide rotated array (originally sorted) into half, either left half or right half must be sorted. To check if the array is sorted, we can simply check if the first element of the array is smaller than the last element of the array. If not, the array is not sorted. Suppose on the contrary, the pivot does not exist within this unsorted array. Then this implies that the pivot exists in the other half of the sorted array. This contradicts the assumption that pivot point must be the local minimum (i.e. elements adjacent to the pivot is greater than the pivot element).

6.

Algorithm

Assume the heap is represented as an array. Then we can find the children from the parent at index i by visiting indices $2i+1$ and $2i+2$.

Extract the minimum

To extract the minimum, simply return the first element in the array. To maintain the heap structure, we will remove the first element in the array as the following. First move the last element in the heap to the head of the heap, and thus overriding the first element. Then we perform an operation called “sink”, where we will keep swapping the head down until it is smaller children. In the worst case, we will need to sink the node to the bottom. Thus, the time complexity of this algorithm is $O(logN)$.

Insert a new number

To insert a new number, simply append the new number to the array (If we need to expand the array, then it will take $O(N)$ time to copy from original array to the new array. But if we are doubling the array size every time it exceeds the capacity, then we can amortize the cost to $O(1)$ time. Thus, we do not need to worry about the case where there is not enough space in the array for time complexity analysis). Then we perform an operation called “swim”. We keep swapping this new number with its parent until the parent is smaller than this new number. In the worst case, we will need to bubble up all the way to the top of the heap. Thus, the time complexity of this algorithm is $O(logN)$.

Change a number

If the number is not changing, then we do nothing. If we are increasing the number, then we perform “sink” operation on that node as described in “Extract the minimum” section above. If we are changing it to a smaller number, then we perform “swim” operation as described in “Insert a new number” section above. Either case, the node travels the height of the heap in the worst case. Thus, the time complexity of this algorithm is $O(logN)$.