Amit Borase

Question 1

Let OPT be the time taken by the optimal solution. Let G be the finishing time of the last job which is nth job.

Claim 1.1

If our greedy algorithm gives a factor c approximation for nice instances, then it gives factor c approximation for all instances.

Proof: Let us assume there exists a non-nice instance of schedule. WLOG, we can assume that such non-nice instance is $d_1, d_2, ..., d_(j-1), d_(j+1), .., d_(k-1), d_(k+1)..., d_n, d_(j), ..., d(k)$, where k,j < n and $d_j >= d_k >= d_n$, such that job n has the maximum finishing-time (last scheduled job doesn't finish last).

Now we employ below exchange argument. Exchange-Argument: In a scheduling, any job that is queued after the last finishing job, does not affect both the G.

Consider a job k which is scheduled after last finishing nth job, then the finishing time of kth job is still smaller than finishing time of nth job. Hence this kth job can be scheduled earlier and it still won't have any impact on G, moreover since job k is scheduled on the machine with the minimum load, it won't change the makespan. Additionally, the OPT will remain the same because it is bounded by the makespan.

Now using above exchange argument, the non-nice scheduling $d_1, d_2, ..., d_(j-1), d_(j+1), .., d_(k-1), d_(k+1)..., d_n, d_(j), ..., d(k)$ can be modified by re-scheduling all $d_k, ..., d_j$ before d_n, to get a scheduling instance $d_1, d_2, ..., d_k, .., d_j, ...d_n$, which is a nice-schedule. Since the nice-schedule has factor c-approximation, the non-nice schedule we started with also has the same factor c-approximation.

Claim 1.2

If $d_n <= OPT/3$ then G <= 4/3.OPT

Proof: We first make couple of claims.

Claim 1.2.1: $OPT >= (d_1 + d_2 + ... + d_n)/m$ [Proved in the class]

Claim 1.2.2: $s <= (d_1 + d_2 + ... + d_n)/m$, where s is the start-time of the last

Now consider the finishing time of the greedy solution, it has to be the start-time of last job plus the duration of last job. Hence

=> G - d_n = s , where s is finishing time of nth job.

=> G - d_n <= $(d_1 + d_2 + ... + d_n)/m$, using claim 1.2.2

=> G <= OPT + d_n, using claim 1.2.1

=> G <= OPT + OPT/3, since $d_n <= OPT/3$

=> G <= (4/3)OPT

=> Hence Proved

Claim 1.3

If $d_n > OPT/3$ then n <= 2m.

Proof: We prove this by contradiction.

Let us assume n > 2m and $d_n > OPT/3$.

Consider the optimal scheduling for n > 2m with OPT as optimal finishing time. Then there exists at-least a single machine m_i (out-of m machines), such that it has more than 2 jobs (at-least 3) scheduled on it.

Let i, j, k be the jobs scheduled on m_i.

Then WLOG, we've d_i, d_j, d_k all with durations times greater than or equal to d_n.

=> d_i, d_j, d_k >= d_n

Now finishing-time of machine m_i = $d_i + d_j + d_k$ >= 3 * d_n

=> finishing-time(m_i) > 3 * (OPT/3), using $d_n > OPT/3$

=> finishing-time(m_i) > OPT

=> This is contradiction because no machine can finish its load after the optimal finishing time.

Hence if $d_n > OPT/3$ then n <= 2m.

Claim 1.4

IF d(n) > OPT/3, then above greedy algorithm gives an optimal solution.

Let us prove this by contradiction. Assume that the above greedy algorithm doesn't give optimal solution. So, there is at least single job k such that job k has finishing time larger than OPT.

From Claim 1.3, we are sure that if $d_n > OPT/3$ then n <= 2m, in other words no machine can have more than two jobs scheduled on it if $d_n > OPT/3$.

let us sort the machines by the number of jobs they have. Then $M_1, M_2, ..., M_i$ be the machines that have only single jobs scheduledon them and $M_i + 1, ..., M_m$ be the machines that has two jobs scheduled on them.

Clearly, all the jobs on the first i machines are going to be longer running jobs than those scheduled on second set of machines, otherwise they could have been able to schedule another jobs on them, but since that is not the case, hence they must be loaded more than the second set of machines.

The kth job that finishes after OPT, is going to be shorter job as well, otherwise it would be needed to be scheduled on the first set of machines (machines with only one job).

Now our greedy algorithm must schedule job k such that it finishes after OPT. But k can not be scheduled on machines running single longer jobs, because it's against greedy-ness of our algorithm. Also k cannot be scheduled with any machines running two short jobs, otherwise that machines will have 3 jobs and which would go against claim 1.3.

in other words, there i machines running i longer jobs and (m - i) machines running shorter jobs. But we've 2(m - i)+1 shorter jobs, (jobs already scheduled and the kth job) to be executed on (m-i) machines each accommodating at-most 2 jobs.

This is clearly not possible. hence our assumption must be wrong.

Hence, greedy algorithm presented above gives optimal schedule if the $d_n > OPT/3$ where d_n is the smallest job and is also the last finishing one.

Question 2

We solve this problem by Dynamic-Programming like algorithm. The strategy is to sort the given data set based on x co-ordinates initially in O(n$log_2 n$) time, using traditional sorting techniques. Now Since, one of co-ordinates is ordered, we only have to compare the y co-ordinate now starting from rightmost points. If it's y co-ordinate is maximum among all y co-ordinates seen so far, then that point is not dominated by any other, because All points to its left have x co-ordinate less than points x co-ordinate and all the points to its right have y co-ordinates less than points y co-ordinate.

Algorithm

1. Sort the given set of points in the decreasing order of their x co-ordinates. If there are points with the same x co-ordinates, then we order them in decreasing order of their y co-ordinates.
2. Now start from the first point $p_1 = (x_1, y_1)$ in the order. This first point is always a point that is not-dominated by anyone else. hence add it to out-list. Adjust y_{max} using $y_{max} = max(y_{max}, y_1)$
3. Move to the next point in the order $p_i = (x_i, y_i)$, if $y_i > y_{max}$, then the point p_i is a point not dominated by anyone else. Adjust y_{max} using $y_{max} = max(y_{max}, y_i)$
4. Continue finding non-dominated points using above step, till we reach the end of the sorted ordering of points.

Proof of Correctness

We prove by induction. We prove the base-case , followed by induction step.

Base-case: The first element $p_1 = (x_1, y_1)$ in sorted set S, is always a non-dominated point.

Proof: Consider that the point $p_1 = (x_1, y_1)$ is not always non-dominated point. In that case there exists at least a single point $p_i = (x_i, y_i)$ in S' such that, it dominates $p_1 = (x_1, y_1)$. According to the definition of non-dominated points:

=> $x_i > x_1$ and $y_i > y_1$

=> X co-ordinate of p_i is greater than that of p_1.

This is contradicting, because then the point p_1 cannot be the first point in the sorted ordering of points, but point p_i will be the new first point in sorted ordering. But p_i is not first point of the sorted ordering.

=> Our assumption is wrong.

=> The point p_1 is always not dominated by any other point in given set of points.

Induction: The point $p_i = (x_i, y_i)$ is a non-dominated point if it's y co-ordinate is greater than y_{max} so far.

Proof: Let us assume that the $y_i > y_{max}$. Then there are below two possibilities regarding dominated nature of point p_i.

First, the point p_i is a non-dominated point, then nothing to prove. We're done.

Second, the point is not a non-dominated point. We prove this by contradiction.

If the point p_i is not a non-dominated point, then there exists at least a single point $p_j = (x_j, y_j)$ such that $x_j > x_i$ and $y_j > y_i$.

Clearly p_j can not be to the left of p_i, because all the points on the left of p_i have x co-ordinates less than or equal to that of p_i

=> Point p_j has to be on the right of the point p_i.

Now if p_j is on the right of p_i, then the y_{max} has to be at least y_j or greater than it, since we would have already visited p_j and updated the y_{max}.

Clearly, that is contradicting our assumption that $y_i > y_{max}$. Since 3 equations $y_i > y_{max}$, $y_{max} >= y_j$ and $y_j > y_i$ cannot be simultaneously true.

Hence our second case that the point p_i is not a non-dominated point is wrong.

Hence, the point $p_i = (x_i, y_i)$ is always a non-dominated point if it's y co-ordinate is greater than y_{max} so far.

Pseudo-code

Listing 1: Question 3

```
INPUT: Set of points S' = $p_i' = (x_i', y_i')$ where i is from 1 to n.
OUTPUT: out-list = List of non-dominated points from given set of points.

FIND_NON_DOMINATED_POINTS(S')
    #Sort the points in S' in decreasing order of their x co-ordinates.
    #If two points have same x co-ordinates then sort them in
    #decreasing order of their y co-ordinates.
    S = Sort(S')
    out-list = []
    y-max = 0
    p = pop(S)
    out-list.append(p)
    y-max = y_coordinate(p)
    while (!S.empty())
       p = pop(S)
       if y_coordinate(p) > y-max:
            out-list.append(p)
            y-max = y_coordinate(p)
    return out-list
```

Time complexity analysis

1. The time taken to sort the S' is O(n$log_2 n$)
2. The loop through 218 to 222 executes n times. and does the constant time work every time. Hence time taken by it is O(n)

Total time taken = O(n$log_2 n$) + O(n) = O(n$log_2 n$)

Total time taken by given algorithm to find non-dominated points in S' is O(n$log_2 n$).

Question 3

We use the divide-conquer strategy to solve this problem by reducing the size of solution space.

Let A and B be the two sorted arrays of size n.

Key Strategy: We first start with A[n/2] and B[n/2], if the A[n/2] == B[n/2], then we've found the nth element in the combined-array to be either A[n/2] or B[n/2]. If the A[n/2] > B[n/2], then B[1] to B[n/2] are all in first n elements of combined-array and A[n/2+1] to A[n] are all after nth element in combined-array. hence they cannot be nth element, hence both these sub-arrays can be ignored to reduce the solution space. We do the same if the A[n/2] < B[n/2], then A[1] to A[n/2] are all in first n elements of combined-array and B[n/2+1] to B[n] are all after nth element in combined-array. hence they cannot be nth element, hence both these sub-arrays can be ignored to reduce the solution space.

Algorithm

Let A' be the sorted array of n' distinct integers.

1. Set s1=0, e1=n, s2=0 and e2=n
2. Find the mid-element a_m of the sorted array A (a_m = A[s1+(e1-s1)/2]), similarly find b_m of the sorted array B (b_m = A[s2+(e2-s2)/2])
3. If a_m = b_m , then either a_m or b_m is the required nth element of the combined sorted array. Stop and return the same.
4. If a_m > b_m , then discard the array A[((e1-s1)/2)+1: (e1-s1)], to get updated array A[0: ((e1-s1)/2-1)]. Also discard the array B[0:((e2-s2)/2)], to get updated array B[((e2-s2)/2):(e2-s2)]. Start again from 1 and work on these updated sub-arrays.
5. If a_m < b_m , then discard the array B[((e2-s2)/2)+1: (e2-s2)], to get updated array B[0: ((e2-s2)/2-1)]. Also discard the array A[0:((e1-s1)/2)], to get updated array A[((e1-s1)/2):(e1-s1)]. Start again from 1 and work on these updated sub-arrays.
6. In case the array-sizes become 1, then manually compare the element a in A and b in B, if a > b or a=b, then a is the nth element. Otherwise B is the nth element.

Proof of correctness

Let A_i and B_i represent the sub-arrays at the ith iteration of above divide-conquer algorithm. Note that index of middle-elements of these arrays will be (n/2^i).

Base-Case: If the A_i >= $B - i$, in i = $log_2 n$ iteration then A_i is the nth element of the combined sorted array.

Proof: Base cases are encountered when we don't find the A_i == $B - i$ till the $log_2 n$ iteration. Till then, we would have removed total $(n/2 + n/4 + ... + n/2^(log_2 N - 1)$ elements that are smaller than nth element of the combined sorted array of A and B.

But $(n/2 + n/4 + ... + n/2^(log_2 N - 1) == (n - 2)$

that means we've already removed (n-2) elements that are to the left of nth element in combined sorted array. hence in base-condition the smallest number will occupy (n-1)th position and the larger element will occupy nth position.

hence Proved.

Induction Step: At any iteration i, if the $A_i[mid1]$ > $B_i[mid2]$ then sub-array $B_i[0 : mid2]$ contains n/2^i elements that come before th enth element in th ecombined sorted array of A and B. **Proof:** We start with 1st iteration.

There are three possibilities:

case1: A[n/2] == B[n/2]

Now if this is the case, then we've exactly, 2* (n/2-1) = (n - 2) elements that are smaller than either of A[n/2] or B[n/2], hence they will occupy the (n - 2) positions before nth element in combined sorted array of A and B. The next element can be either A[n/2] or B[n/2], and hence nth element in the combined sorted array becomes either A[n/2] or B[n/2].

case2: A[n/2] > B[n/2]

If this is the case, we can argue that the elements lesser than B[n/2] in B will always occupy n/2 positions before nth element in the combined sorted array of A and B. Assume one of the element B_i in B[0:n/2] occupies position greater than n. Then we know that B_i < A[n/2] and B_i < B[n/2].

We also know that there are exactly n/2 elements greater than B_i after A[n/2] and B[n/2] to occupy the position in combined array.

This is contradictory. Hence our assumption is worng. hence there cannot be a element in B[0:n/2] such that it takes positions greater than n in sorted combined array.

On similar lines we prove that A[n/2:n] are going to take positions from n+1 to 2n in the sorted combined array of size 2n.

Both these sub-arrays that do not have elements that can be at nth positions can be removed from our search-space, to reduce the search space by 2.

case3: $A_i < B_i$

WLOG, Case3 is indentical to case2. hence can be argued in similar way.

hence, with every iteration we reduce the search-space size by factor of 2.

Generalizing for ith iteration: Total number of elements that occupy index before nth, removed till ith iteration will be: $n/2 + n/4 + ... + n/2^i$

If A[n/2] ==

Hence greedy will be the optimal solution till.

Pseudo-code

Listing 2: Question 3

```
1  INPUT: sorted array's A and B of size n
2  OUTPUT: nth element of the combined sorted array.
3
4  NTH_FIND(A, B, n)
5      s1,s2 = 0
6      e1,e2 = n
7      while ((e1 - s1) >= 1) && ((e2 - s2) >= 1)
8          mid1 = s1+(e1 - s1)/2
9          mid2 = s2+(e2 - s2)/2
10         if A[mid1] == B[mid2]:
11             return A[mid1]
12         else if A[mid1] > B[mid2]:
13             e1 = mid1
14             s2 = mid2+1
15         else if A[mid1] < B[mid2]:
16             s1 = mid1+1
17             e2 = mid2
18
19     if(A[s1] == B[s2]):
20         return A[s1]
21     else if (A[s1] > B[s2]):
22         return A[s1]
23     else if (A[s1] < B[s2]):
24         return B[s2]
```

Time complexity analysis

- At each iteration of the while loop through line 7 to 17, the search space is reduced by factor of 2. Hence while loop executes log_2n times.

- Lines 19 through 24 are the base-case for the divide-and-conquer strategy and they execute in constant time. Hence T(1) = O(1)

We totally divide array log_2n times. Additionally divide step involves comparison of mid-elements $a_m and b_m$, which takes O(1) time.

Hence, the time complexity equation for above algorithm comes out to be:
T(n) = O(1) + T(n/2)
T(n) = O(1) + O(!) + T(n/4)
T(n) = O(1)+ .... + T(1), where T(1) is the log_2n th term in the equation.
T(n) = log_2n * O(1), using T(1) = O(1)
T(n) = O(log_2n)
Hence the running time of above algorithm is of O(log_2n)

Question 4

We use the divide and conquer methodology to solve this problem. WLOG, let us asume that the given array is sorted in increasing order of elements. (If it is sorted in decreasing order then traverse backwords, to get the array on increasing order)

Key Strategy: If the middle-element m of the sorted array is greater than its index then none of the elements following m in sorted array can be same as their respective indices. Similarly if m is less than its index, then none of elements before m in array can have same indices as themselves.

Algorithm

Let A' be the sorted array of n' distinct integers.

1. Set start=0, end=n
2. Find the mid-element a_m of the sorted array A (a_m = A[(end-start)/2])
3. If a_m = ((end-start)/2), then a_m is the required element with same index as itself. Stop and return true.
4. If a_m > ((end-start)/2), then discard the array A[((end-start)/2): (end-start)], to get updated array A[0: ((end-start)/2-1)]. Start again from 1 and work on this updated sub-array (A = A[0, ((end-start)/2-1)]
5. If a_m < ((end-start)/2), then discard the array A[0: ((end-start)/2)], to get updated array array A[((end-start)/2+1): (end-start)]. Start back from 1 and work on this updated sub-array (A = A[((end-start)/2+1): (end-start)]
6. In case the array-size becomes 2, then just manually check if any of the element is equal to its index in A, if it is then return true, otherwise return false.

Proof of correctness

Let i represent the mid-element that is found at every divide step of an algorithm.

Base Case: If A[i] == i, A[i] is the answer.

Proof: Nothing to prove, we already have a element with same value as its index in A.

Induction Step: If A[i] > i, at any step of algorithm. Then none of the elements of the array after i^{th} element can be same as its index in A, i.e. A[j] != j for all j > i.

Proof: We prove this by contradiction. We've A[i] > i.

Let us assume that there exists an element in A[i:length(A)] that is of the same value as its index in A.

WLOG, Let A[j] be such element, then A[j] == j and j > i.

Now there has to be (j-i) number of elements in between ith and jth elements in array A.

Now these (j-i) number of elements can take values that are between A[j] and A[i].

Since sorted array A has all distinct elements, number of distinct values that can be taken up by these (j-i) elements is (A[j] - A[i]) = (j - A[i]) .... using (A[j] == j)

Now number of elements has to be less than or equal to number of distinct values they can take.

hence, (j-i) <= (j - A[i])

This is not possible because LHS is greater than the RHS because A[i] > i.

Hence our assumption must be wrong. hence there cannot exist a element in sorted array A after i^{th} element such that its value is same as its index in A, if A[i] > i

Hence search-space can be reduced by half removing the sub-array A[i:length(A)] from the search-space.

Pseudo-code

Listing 3: Question 3

```
 1  INPUT: sorted array A of size n
 2  OUTPUT: TRUE if A has element such that A[i] == i, else FALSE.
 3
 4  MATCHING_INDEX_FIND(A, n)
 5      start1 = 0
 6      end1 = n
 7      while ((end1 - start1) > 1)
 8          mid = (end1 - start1)/2
 9          if A[mid] == mid:
10              return TRUE
11          else if A[mid] > mid:
12              end1 = mid - 1
13          else if A[mid] < mid:
14              start1 = mid + 1
15      if (A[end1] == end1)
16          return TRUE
17      else if A[start1] == start1
18          return TRUE
19      else
20          return FALSE
```

Time complexity analysis

- At each iteration of the while loop through line 7 to 14, the solution space is reduced by factor of 2. Hence while loop executes log_2n times.
- Lines 15 through 19 are the base-case for the divide-and-conquer strategy and they execute in constant time. Hence T(1) = O(1)

We totally divide array log_2n times. Additionally divide step involves comparison of mid-element which take O(1) time.

Hence, the time complexity equation for above algorithm comes out to be:

T(n) = O(1) + T(n/2)

T(n) = O(1) + O(!) + T(n/4)

T(n) = O(1)+ .... + T(1), where T(1) is the log_2n th term in the equation.

T(n) = log_2n * O(1), using T(1) = O(1)

T(n) = O(log_2n)

Hence the running time of above algorithm is of O(log_2n)

Question 5

This problem can be solved using divide-conquer approach.

Algorithm

1. Sort the given lines in the increasing order of their slopes.
2. Partitions the input into two parts and work on these parts till we reach the base case where input size of sub-problem is 3 or less.
3. If the input size is 3, then add 1st and 3rd line to list of visible line. Calculate the intersection points as well.
4. If the input size is 3, then if the 2nd line intersects 1st line before it intersects 3rd line then add 2nd line as well to list of visible lines. Calculate its intersection point with both the lines.
5. In the merge step, We merge the obtained list visible lines and intersection points as follows:
   - find the uppermost line from left partition and uppermost line from right partition such that left one is intersecting right one.
   -

Proof of correctness

We employ divide-conquer methodology to solve this.

BASE CASE: if the input size is less than 3, then we find the intersection points of 3 lines L1, L2 and L3 with each other. Mote that L1 and L3 will always be visible. L2 is visible only if it intersects L1 before the L3.

Induction Case: IIf the input size is greater than 3, we divide input into partitions recursively till we reach base conditions. In the merge we combined the set of visible lines and intersection points obtained for both th epartition. This can be done by O(n) time.

Pseudo-code

Listing 4: Question 3

```
INPUT: S = {$a_i.x + b_i$, for  i in 1 to n} and n is the input size.
OUTPUT: List of the lines that are visbile in S.
```

```

L=List of lines
n=Number of lines
Sort L in increasing order of their slope

FIND_VISIBLE_LINES(L,n):
    list_lines = {}
    list_IP = {}
    if n <= 3 :
        I1(x1, y1) = find_intersection_point(L[1], L[2])
        I2(x2, y2) = find_intersection_point(L[2], L[3])
        I3(x3, y3) = find_intersection_point(L[3], L[1])
        if x1 < x3:
            list_lines.append(L1,L2,L3)
            list_IP.append(a1,a2)
        else
            list_lines.append(L1,L3)
            list_IP.append(a3)
        return (list_lines, list_IP)
    # If greater than 3 lines, then divide into two partitions and work on them.
    left,left_IP = FIND_VISIBLE_LINES(L[0 : n/2], n/2)
    rightl,right_IP = FIND_VISIBLE_LINE(L[n/2+1 : n], n/2)
    return MERGE_PARTITIONS(left, left_IP, right, right_IP)

MERGE_PARTITIONS(left, left_IP, right, right_IP):
    merge_IP = Merge and sort left_IP and right_IP
    for k in merge_IP

        Merge such that the uppermost line in left is placed above the uppermost ↩
           line in right at x-coordinate cl
        Let (x,y) be the point at which Lis and Ljt intersect.
        merge_lines = Li1,..Lis,Ljt,..,Ljq
        merge_IP = ai1,..ai( s 1 ), (x, y),bjt ,...bj( q 1 )
    return [merge_lines, merge_IP]
```

Time complexity analysis

Merging partitions takes O(n) times at worst.

Hence, Recurrence equation will be:

T(n) = 2T(n/2) + O(n)

Hence the complexity will be O($nlog_2$n) .... Using Master theorem.