Algorithm CS 610 Assignment 2

Naren Kumar S (25310044), Dikshit Hegde (25310017)

04 September 2025

Contents

1 Problem Set 5, Question 09

Given n teams, generate a schedule matches for $n-1$ days, and each team has played a match with all the remaining teams. Here n is strictly in the powers of 2.

1.1 Description

We divide the teams in left half and right half and solve them recurrsively. Left half and right half will give us $\frac{n}{2}-1$ days schedule. For remaining $\frac{n}{2}$ days we pair each team from left half with all the teams in right half in round robin format.

1.2 Algorithm

Algorithm 1 ScheduleMatch(A)

```
1: n ← length of A
2: if n = 2 then
3:    return [(A[1], A[2])]
4: end if
5: leftHalf ← ScheduleMatch(A[1 ... n/2])
6: rightHalf ← ScheduleMatch(A[(n/2 + 1) ... n])
7: finalSchedule ← combine leftHalf with rightHalf for (n/2 − 1) days
8: for i = 1 to n/2 do
9:    dayList ← [ ]
10:   for j = 1 to n/2 do
11:      k ← (i + j) mod (n/2)
12:      append (A[j], A[n/2 + k + 1]) to dayList
13:   end for
14:   append dayList to finalSchedule
15: end for
16: return finalSchedule
```

1.3 Proof of Correctness

Lemma: For a given n teams, each team it should be matched with every other team in such a way that the total number of days should be $n-1$. Match needs to be scheduled in round robin format.

Proof by induction
Base Case: At $n = 2$, the match is scheduled between the 2 teams and its carried out in 1 day [$(n-1)$ days].

Induction Hypothesis:
Assume that the algorithm works correctly for all the teams of size $(\frac{n}{2})$ where n is a 2^x.

Induction Step:
For a given set of n teams, we divide them in 2 halves and solve them recursively. By induction hypothesis, we known that the 2 halves are solves correctly that implies $\frac{n}{2}$ teams matches are scheduled in $\frac{n}{2} - 1$ days. In remaining n days the matches that needed to be scheduled, one from left half and other from right half.

1.4 Time Complexity

The recurrence relation for the above algorithm is $T(n) = 2T(\frac{n}{2}) + n^2$. By using Master Theorem the time complexity of the above algorithm is $O(n^2)$.

2 Problem Set 5, Question 10

Find all undominated points from a given set of n points.

2.1 Description

For a given n points, we first sort the points with respect to x coordinate and divide them in left half and right half. We solve this left half and right half recursively. After solving for individual halves, we consider the undominated points from left half and check if there are any points in right half which dominates the point. This is validated by the MAX of y co-ordinated of undominated points from right Half. If there are no points in right half then the considered point is a undominated point. All the undominated points in the right half are considered as the answer as there are no points in left half which dominated the right half points, since they are sorted along x co-ordinate.

2.2 Algorithm

Algorithm 2 getUndominated(A)

```
1: n ← length of A
2: if n = 1 then
3:    return [A[1]] {Return single element list}
4: end if
5: leftHalf ← getUndominated(A[1...n/2])
6: rightHalf ← getUndominated(A[n/2+1...n])
7: U ← rightHalf
8: y_max ← maxY(rightHalf)
9: for all p in leftHalf do
10:   if p.y ≥ y_max then
11:      append p to U {p is undominated, add to U}
12:   end if
13: end for
14: return U
```

2.3 Proof of Correctness

Lemma: Let leftHalf and rightHalf be the set of undominated points in the left and right half respectively. Every Undominated point is either in rightHalf or leftHalf with $y \geq maxY(rightHalf)$

Proof by induction :

Base Case : $n = 1$
If the set has only one point , by default it is the undominated point.

Induction Hypothesis :
For every set of size smaller than n, the algorithm returns the undominated set correctly.

Induction Step:
For n points we divide the input into two halves , leftHalf and rightHalf .
By the induction hypothesis , leftHalf contains the undominated points present in $A[1 \dots n/2]$ and rightHalf contains the undominated points present in $A[n/2+ 1 \dots n]$.

We observe that, the point in leftHalf can be dominated by the points in rightHalf, so we need to consider a global Undominated set U

- Any undominated point which lies in rightHalf is already a part of U, because points in leftHalf have smaller or equal x-coordinates and cannot dominate points in rightHalf.
- for a point p in leftHalf : if there exist some point q in rightHalf with $q.y \geq p.y$, then q dominates p , so p is not undominated in the whole set. otherwise, if $p.y \geq maxY(rightHalf)$, then it is appended to the undominated set U.

By induction the algorithm correctly finds all the undominated points in the given set.

2.4 Time Complexity

The recurrance relation for the above algorithm is $T(n) = 2T(\frac{n}{2}) + cn$. By using Master Theorem the time complexity of the above algorithm is $O(nlogn)$.

3 Problem Set 5, Question 13

Finding a bad coin from a given set of n coins. Weight of bad coin is slightly less than the good coins

3.1 Description

For a given n coins, we divide them in to exactly two halves and keep the remaining coin in extra. We check the weight of the two halves, if they are equal then the extra contains the bad coin, else the one with less weight will contain the bad coin.

3.2 Algorithm

Algorithm 3 Find(A)

1: $n \leftarrow$ length of A
2: **if** $n = 1$ **then**
3: **return** $A[1]$
4: **end if**
5: $leftHalf \leftarrow$ first $\lfloor n/2 \rfloor$ elements of A {e.g., $A[1 \dots \lfloor n/2 \rfloor]$}
6: $rightHalf \leftarrow$ next $\lfloor n/2 \rfloor$ elements of A {e.g., $A[\lfloor n/2 \rfloor + 1 \dots \lfloor n \rfloor]$}
7: $extra \leftarrow$ remaining elements of A {elements after midHalf}
8: $k \leftarrow$ balance($leftHalf, midHalf$)
9: **if** $k = 2$ **then**
10: **return** Find($extra$)
11: **else if** $k = 1$ **then**
12: **return** Find($rightHalf$)
13: **else**
14: **return** Find($leftHalf$)
15: **end if**

3.3 Proof of Correctness

Lemma : In every comparison step of the algorithm, we use balance to correctly identify a pile of coins that must contain the bad coin, while ignoring all others.

Proof by induction :
Base Case:
In the base case, $n = 1$. Only one coin is there , so we assume it to be the bad coin and return it
Induction Hypothesis :
Assume that the algorithm works correctly for $n - 1$ coins.

Induction Step:
We have a pile of n coins.

- Case 1 : If n is even
  If n is even, the algorithm splits pile of coins into two equal halves and extra will be 0.
  With balance function we determine whether the bad coin is present in the first half , second half, or in neither of them,
  By the lemma, only one of the half can contain the bad coin.
  The algorithm recursively solve the half which contains the bad coin.
  By the Induction Hypothesis,we know that the recursive call will correctly identify the bad coin.

- Case 2 : If n is odd
  If n is odd, the algorithm splits the pile of coin into two equal halves and the single coin is stored in extra.
  With balance function we determine which pile the bad coin is present and choose the pile which is less in weight.If they are equal then extra contains the bad coin.
  Again by lemma, the chosen pile will contain the bad coin.
  We search the chosen pile recursively and get the bad coin, supported by the Induction hypothesis.

3.4 Time Complexity

The recurrence relation for the above algorithm is $T(n) = T(\frac{n}{2}) + c$. Based on the master theorem and solving the recurrence relation, the time complexity of the above algorithm is $O(logn)$

4 Problem Set 5, Question 17

Given a $n \times n$ matrix A, where each rows are sorted in ascending orders and each columns are also sorted in ascending order. Find a query Q in matrix A.

4.1 Description

For a given matrix A whose rows and columns are sorted in ascending order. For a given query Q, we first check the top right element ie. $A[1][n]$ with the query. If the query Q is less than the top right element, then that particular column doesnt contain that Query, so we reduce the index by 1 which is iterated over columns. If the query Q is greater than the top right element then that particular row doesnt contain the query, so we increase the index by 1 which is iterated over rows. If the indexing is at bottom left corner then the query is not present in the A matrix.

4.2 Algorithm

Algorithm 4 Find(A, Q)

```
1: (n, m) ← shape of A
2: if n = 1 and m = 1 then
3:    if A[1][1] = Q then
4:       return True
5:    else
6:       return False
7:    end if
8: end if
9: i ← 1
10: j ← m
11: while i ≤ n and j ≥ 1 do
12:    if A[i][j] = Q then
13:       return True
14:    else if A[i][j] > Q then
15:       i ← i + 1
16:    else
17:       j ← j − 1
18:    end if
19: end while
20: return False
```

4.3 Proof of Correctness

Lemma : Let A be a matrix of size $n \times n$. At each step, the algorithm matches the element $A[i][j]$ with Q. If $A[i][j] > Q$, then Q cannot lie in column j and

if $A[i][j] <$ then Q cannot lie in row i.

Proof by induction :
Base Case: In base case, $n = 1$. we get a matrix of a single element. If $A[1][1] = Q$, then the algorithm finds it. Otherwise, it returns false.

Induction Hypothesis :
Assume that the algorithm works correctly for all the matrices of size $(n-1) \times (n-1)$ with the sorted row and column property.

Induction Step :
The size of the matrix A is $n \times n$. We start at the top right of A i.e $A[1][n]$:

- Case 1:
  If $A[1][n] = Q$, we found the element.

- Case 2:
  If $A[1][n] > Q$, then by lemma , Q cannot be in the column n .
  We eliminate the column n and the problem reduces to $n \times (n-1)$ matrix, by induction the algorithm works on this reduced matrix.

- Case 3:
  If $A[1][n] < Q$, then by lemma, x cannot be in the row 0.
  We eliminate that row and reduce the matrix to the size $(n-1) \times n$, by induction the algorithm works on this reduced matrix.

- We observe that in either case the algorithm reduces the matrix to a smaller instance i.e $(n-1) \times (n-1)$

By induction the algorithm correctly finds whether the value Q is present in the matrix A or not.

4.4 Time Complexity

In the above algorithm, the time complexity depends on the while loop where the loop runs for $n+m$ times. The Time complexity of the algorithm is $O(n+m)$ approximately $O(n)$.