Analysis and Design of Algorithms

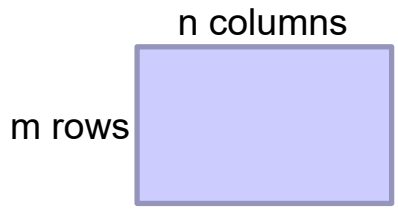
Algorithms

CS3230

Tutorial

Week 4

Question 1



Suppose we are given a 2D-array A of size m rows by n columns. An element in the array $A[i][j]$ is called a "peak" if it is **greater than or equal to** its adjacent neighbours (if they exist -- an element is always considered greater than or equal to non-existent elements). For example, the 8 in the middle is a peak in the following array.

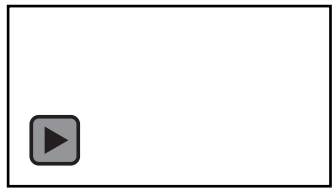
```
*  *  5  *
*  8  8  3
*  *  2  *
```

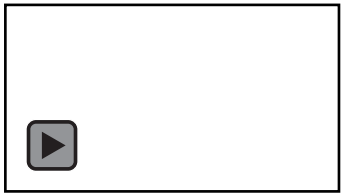
Consider the following algorithm to return any "peak":

```
Find2DPeak(A):
    If A only has a column, return the maximal element of the column
    Otherwise:
        Select the middle column of the A
        Find the maximal element of the column
        If the maximal element is a peak, return that element
        Else
            p1 = Find2DPeak(right half of A excluding middle col)
            p2 = Find2DPeak(left half of A excluding middle col)
            If p1 or p2 is a peak, return either one, otherwise return None
```

What is the runtime of the algorithm?

- $\Theta(mn)$
- $\Theta(m \lg n)$
- $\Theta(\lg m \lg n)$
- $\Theta(m^2 \lg n)$





Question 1

Answer: A

Consider the *number of times a column is processed*.

- In the divide step, the middle column is processed, and no column is processed in the combined step. $O(1)$
- The number of columns is halved in the recursive step
- Two subproblems are solved.

Question 1

The recurrence for the number of times a column is processed is
$T(n) = 2T(n/2) + 1$

To solve using the Master theorem, we check
f(n) = $\Theta(1) = O(n^{\lg 2 - \varepsilon})$ giving
$T(n) = \Theta(n)$ by case 1.

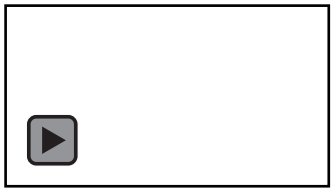
And the time to process a column is $\Theta(m)$ so total time is $\Theta(mn)$.

Question 2

Suppose we are given a 2D-array A of size *m* rows by *n* columns. An element in the array $A[i][j]$ is called a "peak" if it is **greater than or equal to** its adjacent neighbours (if they exist -- an element is always considered greater than or equal to non-existent elements). For example, the 8 in the middle is a peak in the following array.

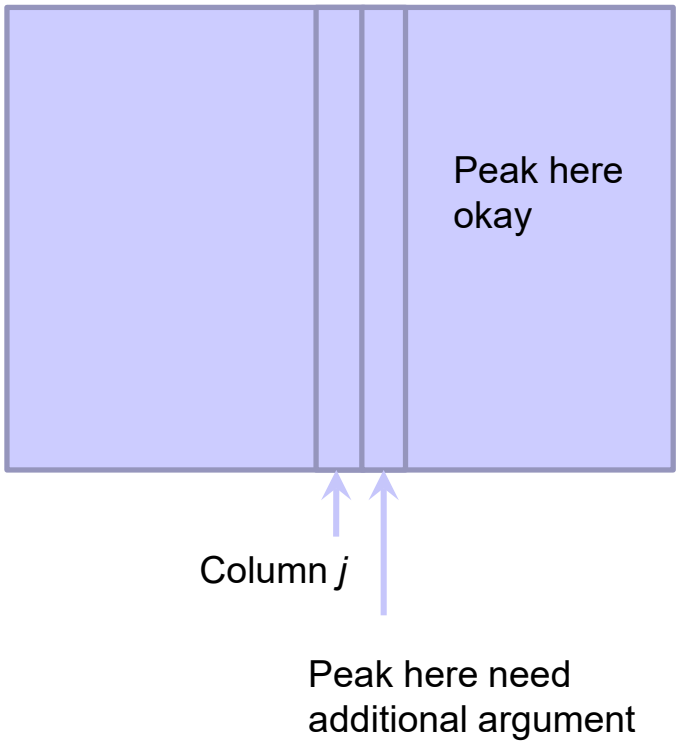
$$\begin{matrix} * & * & 5 & * \\ * & 8 & 8 & 3 \\ * & * & 2 & * \end{matrix}$$

Let *A*[*i*][*j*] be the largest element in column *j*. Assume that $A[i][j+1] \geq A[i][j]$. Argue that any peak in the the subarray *A*[1..*m*][*j*+1..*n*] that is the largest element in its column is also a peak of the entire array *A*.



Answer

- Note that the largest element in A[1..m][j+1..n] is a peak in A[1..m][j+1..n].
- If the peak is not located on column *j+1*, it is certainly a peak in *A*.
- Need more argument if the largest element is located on column *j+1*.



Answer

Let the largest element in column j be $c_{i,j}$.

Assume $c_{k,j+1}$ is a peak in $A[1..m][j+1..n]$ and is the largest element in column j+1.

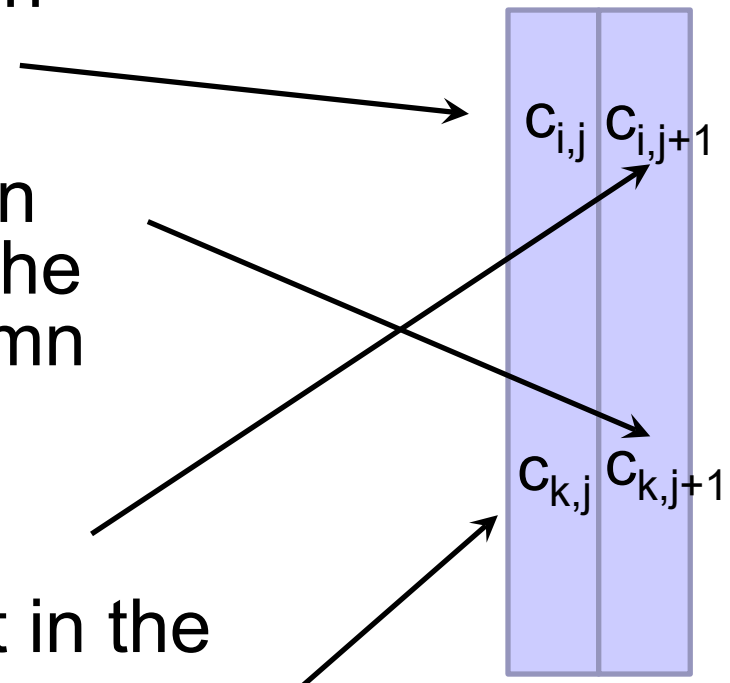
Then

$c_{k,j+1} \geq c_{i,j+1}$ ($c_{k,j+1}$ largest in the column)

$\geq c_{i,j}$ (by assumption)

$\geq c_{k,j}$ ($c_{i,j}$ largest in column)

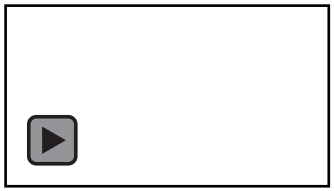
So $c_{k,j+1}$ is a peak in A.



Question 3

Suppose we are given a 2D-array A of size *m* rows by *n* columns. An element in the array $A[i][j]$ is called a "peak" if it is **greater than or equal to** its adjacent neighbours (if they exist -- an element is always considered greater than or equal to non-existent elements). For example, the 8 in the middle is a peak in the following array.

$$\begin{matrix} * & * & 5 & * \\ * & 8 & 8 & 3 \\ * & * & 2 & * \end{matrix}$$



Using the idea in Question 2, describe an algorithm that is asymptotically faster that the one given in Question 1. What is it's runtime?

Answer

Idea: Using Q2, reduce the number of subproblems in Q1 from 2 to 1

```
Find2DPeak(A):

    If A only has a column, return the maximal element of the column

    Otherwise:

        Select the middle column of the A

        Find the maximal element of the column

        If the maximal element is a peak, return that element

        Else if the maximal element is smaller than the element on its right

            Recurse on right half of A (excluding middle column)

        Else recurse on left half of A (excluding middle column)
```

Answer

The recurrence for the number of times a column is processed is
$T(n) = T(n/2) + 1$

To solve using the Master theorem, we check
f(n) = $\Theta(1) = \Theta(n^{\lg 1} \log^0 n)$ giving
$T(n) = \Theta(\lg n)$ by case 2.

And the time to process a column is $\Theta(m)$ so total time is $\Theta(m \lg n)$ (previously $\Theta(mn)$).

Can use Q2 with mathematical induction to argue correctness.

Question 4

FIB'(*n*)

1. **if** *n* == 0
2. **return** 0
3. **elseif** *n* == 1
4. **return** 1
5. *sum* = 1
6. **for** *k* = 1 to *n* - 2
7. *sum* = *sum* + FIB'(*k*)
8. **return** *sum*

Your clever classmate suggests the following way to compute the Fibonacci number.

It is not obvious that FIB'(n) correctly returns the n-th Fibonacci number. To understand the algorithm, you do a correctness proof of the algorithm.

The proof will be done by induction. For the base cases of n = 0 and n = 1, the algorithm simply returns the value defined, hence is correct. As inductive hypothesis, we assume that FIB'(k) correctly return the k-th Fibonacci number for all k < n.

Assuming the inductive hypothesis, we need show that FIB'(n) works correctly. Within FIB'(n) is a loop (line 6 and 7), and we need to show that the loop correctly returns the n-th Fibonacci number assuming that FIB'(k) correctly return the k-th Fibonacci number for all k < n (from the inductive hypothesis).

State a suitable loop invariant (line 6 and 7). Then, show that FIB'(n) works correctly

Solution

A suitable invariant is "*sum* = F_{k+1}"

sum = 1
for *k* = 1 **to** *n* - 2
  sum = *sum* + FIB'(*k*)

Invariant

$$\text{sum} = F_{k+1}$$

Initialization:

sum = 1, *k* = 1,

Before entering loop, sum = 1 = F_2 as required.

sum = 1
for *k* = 1 **to** *n* - 2
  sum = *sum* + FIB'(*k*)

Invariant

$\text{sum} = F_{k+1}$

Maintenance: Assume invariant true at start of iteration. Show invariant true at the start of next iteration.

Within iteration variable *sum* becomes
  sum' = *sum* + FIB'(*k*) = $F_{k+1} + F_k = F_{k+2}$
k becomes *k'* = *k*+1, so *sum'* = $F_{k+2} = F_{k'+1}$

sum = 1
for *k* = 1 **to** *n* - 2
 sum = sum + FIB'(*k*)

Invariant

$\text{sum} = F_{k+1}$

Termination: *k* = *n*-1, so *sum* = F_n

Together with assumption that FIB'(*k*) = F_k (inductive hypothesis for proof of recursive algorithm) we have shown that FIB'(*n*) = F_n for all *n*.

FIB'(*n*)

1. **if** *n* == 0
2. **return** 0
3. **elseif** *n* == 1
4. **return** 1
5. *sum* = 1
6. **for** *k* = 1 to *n* - 2
7. *sum* = *sum* + FIB'(*k*)
8. **return** *sum*

To prove FIB'(*n*) returns F_n correctly:

Base case:

FIB'(0) return 0, which is F_0.
FIB'(1) returns 1, which is F_1.

Hence, base cases are true.

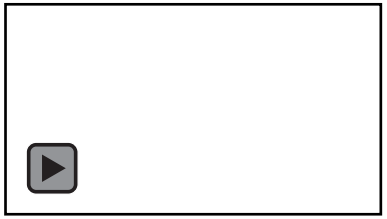
Inductive step: Assume FIB'(*k*) correct for $k < n$. Invariant shows FIB'(*n*) returns F_n upon termination.

Question 5

FIB'(*n*)

```
1. if n == 0
2.     then return 0
3. elseif n == 1
4.     then return 1
5. sum = 1
6. for k = 1 to n-2
7.     do sum = sum + FIB'(k)
8. return sum
```

Can you give the recursive formula for the runtime of this algorithm?



Answer

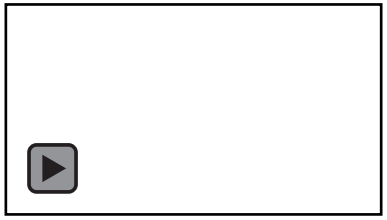
The “for” loop gives 1+FIB’(1)+…+FIB’(n-2)
giving the recursive calls.
Looping n-2 time takes time proportional to n

Hence
$T(n) = T(1) + \ldots + T(n-2) + a \cdot n$

Question 6

For the recursive formula in the previous question, can you analyze its time complexity?

(Hint: You can check $T(n) - T(n-1)$.)



Answer

$T(n) = T(1) + \ldots + T(n-3) + T(n-2) + an$
$T(n-1) = T(1) + \ldots + T(n-3) + a(n-1)$

$T(n) - T(n-1) = T(n-2) + a$
$T(n) = T(n-2) + T(n-1) + a$

In Lecture02, this recurrence has been shown to be $T(n) = \Theta(F_n)$ by substitution method.