Problem 2. Young Tableaus.

We define element access for the matrix M as M[(i,j)] – it takes the tuple (i,j) as the input and returns the element M[i][j]

We also define the following procedures that return an element's vertical or horizontal neighbours in the matrix.

```
TOP(M, (i,j))
    if i > 1 return M[(i-1, j)]
    else return NIL

BOTTOM(M, (i,j))
    if i < M.rows return M[(i+1, j)]
    else return NIL

RIGHT(M, (i,j))
    if j < M.cols return M[(i, j+1)]
    else return NIL

LEFT(M, (i,j))
    if j > 1 return M[(i, j-1)]
    else return NIL
```

All the above procedures take $\Theta(1)$ time. (Note that indexing starts at 1)

We can then define the tableau property as:

$$M[(i,j)] \leq \text{BOTTOM}(M,(i,j)) \quad \forall \ 1 \leq i \leq M.\text{rows}$$
$$\text{and } M[(i,j)] \leq \text{RIGHT}(M,(i,j)) \quad \forall \ 1 \leq j \leq M.\text{cols}.$$

This is evident from the definition of a Young tableau.

Here we define an algorithm similiar to MIN-HEAPIFY that restores the tableau property for an element M[(i,j)] by exchanging it with the smaller of its bottom and right neighbour and then restoring the property for that neighbour element. It does nothing if the tableau property is already satisfied.

```
TABLEAUFY(M, (i,j))
1  b = BOTTOM(M, (i,j))
2  r = RIGHT(M, (i,j))
3  curr = M[(i,j)]
4  if b != NIL && curr > b
5      smallest = (i+1, j)
6  else smallest = (i,j)
7  if r != NIL && curr > r
8      smallest = (i, j+1)
9  if smallest != (i,j)
10     M[(i,j)] <-> M[smallest]
11     TABLEAUFY(M, smallest)
```

Proof of Correctness: By exchanging $M[i,j]$ with $\min(M[(i+1,j)], M[(i,j+1)])$ we ensure that both $M[(i,j)] < \text{BOTTOM}[M(i,j')]$ and $M[(i,j)] < \text{RIGHT}(M,(i,j))$ are satisfied. Thus TABLEAUFY restores the tableau property for $M[i,j]$ and then proceeds to do the same for the exchanged neighbour, hence restoring the property to all elements.

Time Complexity Analysis: All the steps involved in finding the smallest of the three elements and exchanging takes $\Theta(1)$ time. Thus we can define the relation for time taken by TABLEAUFY$(M,(i,j))$ as

$$T(i,j) = \Theta(1) + T(\text{smallest}).$$

where smallest can be $(i+1, j)$ or $(i, j+1)$. Further, we have

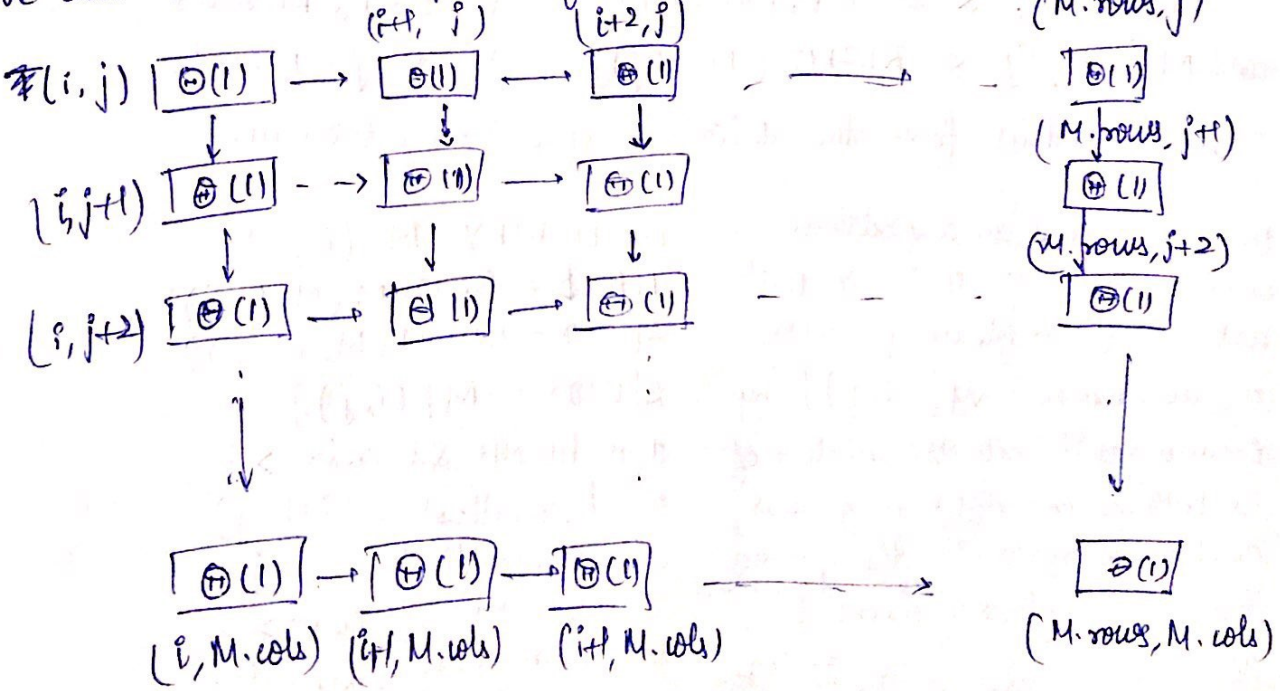
$$T(i,j) \leq \Theta(1) + T(i+1, j) + T(i, j+1).$$

since times are non-negative. Note that

$$T(M.\text{rows}, M.\text{cols}) = \Theta(1)$$
$$T(M.\text{rows}, j) = \Theta(1) + T(M.\text{rows}, j+1)$$
$$T(i, M.\text{cols}) = \Theta(1) + T(i+1, M.\text{cols})$$

since if j cannot exceed $M.\text{rows}$ & $M.\text{cols}$ respectively.

We can construct the following recursion "tree".



Thus in the worst case the recursion ends at $(M.\text{rows}, M.\text{cols})$. Further (i,j) goes to either $(i+1, j)$ or $(i, j+1)$ but never both

Hence we have

$$T(i,j) \leq \Theta(1)(M.rows - i + M.cols - j)$$

Since we require $(M.rows - i)$ right steps & $(M.cols - j)$ down steps to reach $(M.rows, M.cols)$ from (i, j) irrespective of the path taken.

For $m \times n$ matrix, $M.rows = m$ & $M.cols = n$.

So $T(i,j) \leq \Theta(1)(M - i + n - j) = O(m+n)$ $\because i, j \leq m, n$.

Hence running time of TABLEAUFY is $O(m+n)$.

1] As EXTRACT-MIN is defined for a min-heap using MIN-HEAPIFY, we analagous define an EXTRACT-MIN algorithm for a Young tableau using TABLEAUFY.

```
EXTRACT-MIN (M)
1 | value = M[(1,1)] ; M[(1,1)] = ∞
2 | TABLEAUFY (M, (1,1))
3 | return value
```

Proof of correctness:

Note that according to the tableau property, the smallest element of the matrix would be the top left corner element i.e. $M[(1,1)]$

We store this value then replace it with ∞ which represents a non-existant value as mentioned in the definition. As the property is broken for $M[(1,1)]$ now we restore it with TABLEAUFY. Then we return the value of $M[(1,1)]$ stored.

Time Complexity Analysis:

Storing the value of $M[(1,1)]$ and resetting it takes time $\Theta(1)$
Restoring the tableau property takes $O(m+n)$ as proved above.
Hence running time for EXTRACT-MIN is $O(m+n)$.

2] Similar to the insert procedure for heaps, we will follow a 'float-up' approach. Since the tableau is non-full, $M[(M.rows, M.cols)] = \infty$. is guaranteed. So we add the new element at the bottom right corner and then continuously swap it with its top & left neighbours until the tableau property is restored.

```
INSERT(M, key)
1 | i = M.rows, j = M.cols
2 | M[(i,j)] = key.
3 | while (i>1 && j>1) and (M[(i,j)] < TOP(M,(i,j)) || M[(i,j)] < LEFT(M,(i,j)))
4 |   if TOP(M(i,j)) > LEFT(M,(i,j))
5 |     largest = (i-1, j)
6 |   else largest = (i, j-1)
7 |   exchange M[(i,j)] with M[largest]
8 |   (i,j) = largest
  if (j==1)
    while i>1 and M[(i,j)] < TOP(M(i,j))
      exchange M[(i,j)] with TOP(M,(i,j))
      i--;
  if (i==1)
    while j>1 and M[(i,j)] < LEFT(M,(i,j))
      exchange M[(i,j)] with LEFT(M,(i,j))
      j--;
```

Proof of Correctness: In the INSERT algorithm we repeatedly swap $M[(i,j)]$ with $\max(M[(i-1,j)], M[(i,j-1)])$. Consider it was swapped with $M[(i-1,j)]$. Then for $M[(i-1,j)]$ the tableau property is established as now $M[(i,j)] > M[(i-1,j)]$ and we already had $M[(i-1,j+1)] > M[(i-1,j)]$. Thus INSERT puts the new value at a position where tableau property is satisfied

3] In a manner analogous to HEAPSORT, where we repeated heapify the array and then extract the root value to sort the array. We can use the definition of the Young tableau to sort the n^2 input numbers. Following is the pseudocode for the algorithm:

// A is an array of n^2 numbers.

```
TABLEAU_SORT(A).
1 | Let M[n][n] be a Young tableau initialized with ∞.
2 | for i = 1 to n^2
3 |     INSERT(M, A[i])
4 | for i = 1 to n^2
5 |     A[i] = EXTRACT_MIN(M)
```

Proof of Correctness: We insert the elements into the tableau one by one. The insert function maintains the tableau property. Once the tableau is complete, we repeated ly extract M[(1,1)]. EXTRACT_MIN removes the current minimum and then restores the tableau property bringing the next smallest element to M[(1,1)]. Thus above algorithm sorts A.

Time complexity Analyses: A call to INSERT and EXTRACT_MIN each takes $O(m+n)$ time. Here $m = n$ so time is $O(n)$. There are n^2 calls each to INSERT & EXTRACT_MIN.

Hence total time $T(n) = n^2 O(n) + n^2 O(n) = 2n^2 O(n)$

$\therefore T(n) = O(n^3)$. ($\because 2n^2 O(n) = 2n^2 c_1 n = 2c_1 n^3 < 4c_1 n^3 = O(n^3)$)

Hence running time of TABLEAU_SORT is $O(n^3)$.

Time Complexity Analysis : As in the case of TABLEAUFY, from $M[i,j]$ we can either go to $M[(i-1,j)]$ or $M[(i,j-1)]$ or terminate. Thus we start from $M[(M.rows, M.cols)]$ and head towards $M[(1,1)]$. To reach $M[(i,j)]$ from $M[(M.rows, M.cols)]$ we would require $(M.rows - i)$ up & $(M.cols - j)$ left steps. In each step, we spend only $\Theta(1)$ time in finding the largest & exchanging elements.

Hence $T(i,j) \leq \Theta(1)(M.rows - i + M.cols - j)$.

For an $m \times n$ matrix,

$$T(i,j) \leq \Theta(1)(m - i + n - j) = O(m+n) \quad \because \begin{matrix} i \leq m \\ j \leq n \end{matrix}$$

Thus INSERT takes $O(m+n)$ running time.

4] We use the definition of the Young tableau - each column & row is sorted in increasing order. This implies that the values decrease upon moving up a column and the values increase by moving ~~up~~ ~~left~~ right in a row. Thus by these two operations we can reach any element starting from the bottom-left corner. Pseudocode for the algorithm is as follows:

```
TABLEAU_SEARCH ( M, key ).
1 | i = M.rows, j = 1
2 | while ( i ≥ 1 and j ≤ M.cols)
3 |     if M[(i,j)] == key return (i,j)
4 |     else if M[(i,j)] < key j++
5 |     else i--   // M[(i,j)] > key.
6 | return NIL
```

Proof of Correctness: We start at M[(M.rows, 1)] i.e the bottom left corner. If the current value equals the key we return the indices and we are done. However, if the current value is less than the key, we move a column right. Since the values increase on that side the key must ~~be~~ be to the left of current column. If the current value is greater than the key, we move a row up since elements decrease in that direction. Since the values decrease in that way the key must be a in a row above the current row.

Time complexity Analysis: Worst case would be when the loop terminates fully, i.e. the key is not found & NIL is returned. Here would have traversed the whole matrix using right & up steps. till we reached the top left corner. This would take m up steps & n right steps. Since all elements either lie ~~left~~ or below the top left element,

they will definitely require less than or equal to m up steps & less than or equal to n steps to reach from the bottom right corner.

So, we have that

$$T(m, n) \leq (m+n)\,\Theta(1)$$

Since comparing the current element with the key and changing the indices are all $\Theta(1)$ operations.

Hence we have $T(m+n) = O(m+n)$.

Thus TABLEAU_SEARCH takes $O(m+n)$ running time.

ASHWIN SHENAI
180156

HOMEWORK 2

ES0207A
03/09/2019

Problem 1. Heaps.

1] BUILD_MAX_HEAP(A, n).

```
A.heapsize = 1
for i = 2 to n
    MAX-HEAP_INSERT(A, A[i])
```

1] MAX_HEAP_INSERT implements a 'float-up' approach. It makes the new element a leaf node and then recursively pushes it up the tree by swapping it with its parent node until it reaches the correct place.

Let the height of the heap be h.

Then inserting a leaf node into the heap would mean at most that it has to travel height h at most to reach its correct place.

There are $\frac{n}{2}$ leaf nodes, assuming n is a power of 2.

Thus assuming that pushing a node down a level takes $\Theta(1)$ time, pushing the leaf nodes into the heap would take $\frac{hn}{2}\Theta(1)$ time. Similiarly, a node supposed to be at a height $(h-k)$ in the built heap has to travel at most that distance when pushed down. There are $\frac{n}{2^{k+1}}$ nodes at the height $(h-k)$. Thus assuming constant time for pushing a node down a level, building the heap by insertion would take

$$T(n) = \sum_{k=0}^{h} (h-k)\frac{n}{2^{k+1}}\Theta(1)$$

Thus $T(n) = \left(\frac{hn}{2} \left[\sum_{k=0}^{h} \frac{1}{2^{k-1}} \right] - \frac{n}{2} \left[\sum_{k=0}^{h} \frac{k}{2^k} \right] \right) \Theta(1)$

$T(n) \leq \left(\frac{hn}{2} \cdot 2 - \frac{n}{2} \cdot 2 \right) \Theta(1)$

Using $\sum_{k=0}^{h} \frac{1}{2^k} < \sum_{k=0}^{\infty} 1/2^k = 2$ & $\sum_{k=0}^{n} \frac{k}{2^k} \leq \sum_{k=0}^{\infty} k/2^k = 2$.

Thus $T(n) \leq (h-1)\, n\, \Theta(1)$.

Note that $h = \log n$ for a heap.

Hence $T(n) \leq (\log n - 1)\, n\, \Theta(1) = O(n \log n)$

Since $O(n) = O(n \log n)$.

Thus BUILD_MAX_HEAP using MAX_HEAP_INSERT takes $O(n \log n)$ running time.

2] Note that MAX-HEAP-INSERT pushes an element from the bottom of the heap to its correct location. Thus for an input array $A[1 \ldots n]$ sorted in increasing order, every element $A[i]$ must be pushed to the top (root) of the heap since it is a max heap & $A[i]$ will be greater than all the elements before it.

Therefore, for every element $A[i]$, we push it $\log_2(i)$ levels, which is the height of the heap with $(i-1)$ levels

Hence total time taken for this worst case input is

$$T(n) = \sum_{i=2}^{n} c \log_2(i)$$ where c is the time for comparisons & exchanges.

$$= \sum_{i=1}^{n} c \log_2(i) .$$

Considering only the second half of the series we can say

$$T(n) \geq \sum_{i=n/2+1}^{n} c \log_2(i).$$ Since $i \geq n/2+1$, $\log_2 i > \log_2 n/2$

$$\therefore T(n) \geq \sum_{i=n/2+1}^{n} c \log_2(n/2) = c \frac{n}{2} \log \frac{n}{2}$$

$$\therefore T(n) \geq \frac{c}{2}(n \log n - n \log 2) = c' n \log n - c' n \log 2$$

Hence $T(n) = \Omega(n \log n)$ for worst case input.

Since $T(n) = O(n \log n)$ also, we have $T(n) = \Theta(n \log n)$

ASHWIN SHENAI
180156

HOMEWORK 2

ESO207A
03/09/2019

Problem 3. Selection

Firstly, let us define PARTITION (A, p, r, pivot) which returns a tuple (q, s) where q and s are indices that divide input array A into three parts such that

$A[p \ldots q-1] < pivot$, $A[q \ldots s] = pivot$ & $A[s+1 \ldots r] > pivot$.

the function returns the input array in this state.

Pseudocode for the algorithm is as follows:

```
PARTITION ( A, p, r, pivot )
    q = p-1 ; s = p-1 ; // start before first element.
    for j = p to r-1
        if (A[j] == pivot)
            exchange A[s+1] and A[j];
            s++;
        else if ( A[j] < pivot )
            exchange A[j] with A[s+1];
            exchange A[s+1] and A[q+1];
            q++; s++;
        else  // A[j] > pivot, do nothing
    exchange A[r] and A[s+1]
    return (q+1, s+1)
```

Proof of correctness.: PARTITION makes one pass through the array. If it finds an element equal to pivot, it puts it after A[s] and increments s. If it finds an element less than the pivot, it sends it to A[q+1] while shifting A[q+1 .. s+1] to A[q+2 .. s+2]. Then (q+1, s+1) are finally the required indices which split the array as required.

Time Complexity of PARTITION : Since the comparisons and exchanging of elements takes $\Theta(1)$ time, we have

$$T(p,r) = \Theta(1)(q-p+1) = \Theta(r-p+1)$$

We can use the partition function to output all the elements smaller than the pivot in linear time by simply printing $A[p..q-1]$. Thus if we make the k^{th} smallest element the pivot we can return the first k smallest elements.

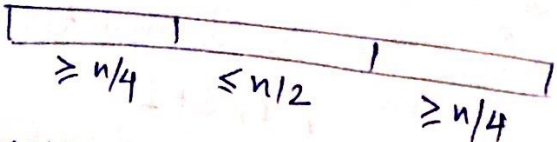
To find the k^{th} smallest element first, we use a randomized version of PARTITION defined as follows:

```
RANDOMIZED_PARTITION (A, p, r).
    i = RANDOM(p, r); n = A.length;
    (q, s) = PARTITION (A, p, r, A[i]).
    if ( q - p ≤ 3n/4 and r - s ≤ 3n/4 ).
        return (q, s)
    else return RANDOMIZED_PARTITION(A, p, r).
```

Our goal here is to choose a pivot that makes the partition as even as possible.

$q - p \le 3n/4$, $r - s \le 3n/4$ $\Rightarrow$ $q - s + r - p \le 3n/2$

$\Rightarrow$ $q - s \le n/2$. $\Rightarrow$



Randomized partition uses RANDOM (p, r) which randomly returns a number between p & r with equal chance. Until we do not achieve the partition we require it is recursively called again & again.

Note that there are (k-1) elements smaller than the k^{th} smallest element. So in a partitioned array, if we have $q - p \geq k$, it implies that the pivot is greater than the k^{th} smallest element and the k^{th} smallest element lies in $A[p \ldots q-1]$.

If we have instead that $k \geq (s+1)$, we have that the pivot is less than the k^{th} smallest element and hence it lies in $A[s+1 \ldots r]$.

If none of these cases hold, we have that k^{th} smallest element must lie in $A[q \ldots s]$. Since all of these are equal we have found the k^{th} smallest element.

Pseudocode for algorithm to find k^{th} smallest element's index as explained above:

```
SELECT_K(A, p, r, k)   // k ≤ r-p+1
    (q, s) = RANDOMIZED_PARTITION(A, p, r);
    if k ≤ q-p
        return SELECT_K(A, p, q-1, k);
    else if k ≥ s-p+2
        return SELECT_K(A, s+1, r, k-(s-p+1));
    else return q.
```

In the second case we must subtract $(s-p+1)$ elements from k since they would not be a part of the active array, as they lie to the left of $A[s+1 \ldots r]$

Then our algorithm to find the k^{th} smallest elements becomes,

```
K_SMALLEST_ELEMENTS (A, k)
    n = A.length;
    j = SELECT_K (A, 1, n, k) . // index of k^th smallest element
(q,s) = PARTITION ( A, 1, n, A[j] )
    return A[1 . . . q-1]
```

Time Complexity Analysis: For randomized partition function, let T be the number of recursive calls before some value is returned.

Then we have $E(T) = 1 + \frac{1}{2} E(T) \Rightarrow \boxed{E(T) = 2}$

where E is expectation.

Running time for RANDOMIZED-PARTITION is $T_1(n) = T * \Theta(n)$ since $\Theta(n)$ is running time for PARTITION.

Hence expected running time is: $E(T_1(n)) = \Theta(n) E(T)$.

$\Rightarrow E(T_1(n)) = \Theta(n)$.

Now for SELECT_K, running time is.

$T(n) = T(\text{'partition'}) + T_1(n)$.

where T('partition') is the time for SELECT_K to run on either of $A[p \ldots q-1]$ or $A[s+1, r]$. As proved earlier, using RANDOMIZED_PARTITION we can prove that the length of these partitions is definitely less than $\frac{3n}{4}$

Thus we get

$$T_2(n) \leq T_2\left(\frac{3n}{4}\right) + T_1(n)$$

expected running time,

$$E[T_2(n)] \leq E\left[T_2\left(\frac{3n}{4}\right)\right] + \Theta(n).$$

Solving we get

$$E[T_2(n)] = \Theta(n)$$

For K_SMALLEST_ELEMENTS,

$$T(n) = T_2(n) + \Theta(n).$$

Hence expected running time for 'K_SMALLEST_ELEMENTS' is $\Theta(n)$.