CSE3358 Problem Set 5
Solution

Problem 1: Young tableau

An $m \times n$ Young tableau is an $m \times n$ matrix such that the entries of each row are in sorted order from left to right and the entries of each column are in sorted order from top to bottom. Some of the entries of a Young tableau may be ∞, which we treat as nonexistent elements. Thus a Young tableau can be used to hold $r \leq mn$ numbers.

Here's an example of a 4x4 Young tableau containing the elements $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$. Note that this is not unique.

2	4	9	∞
3	8	16	∞
5	14	∞	∞
12	∞	∞	∞

(a) (5 points) Argue that an $m \times n$ Young tableau Y is empty if $Y[1,1] = \infty$. Argue that Y is full (contains mn elements) if $Y[m,n] < \infty$.

ANSWER: Assume $Y[1,1] = \infty$. consider $Y[i,j]$ for any i and j. By the property of a Young tableau, $Y[i,j] \geq Y[1,j] \geq Y[1,1]$. Therefore, $Y[i,j] = \infty$. This means that the Young tableau is empty because the above is true for every i and j.

Now assume $Y[m,n] < \infty$. Consider $Y[i,j]$ for any i and j. By the property of a Young tableau, $Y[i,j] \leq Y[m,j] \leq Y[m,n]$. Therefore, $Y[i,j] < \infty$. This means that the Young tableau is full because the above is true for every i and j.

(b) (5 points) Argue that the minimum element of a Young tableau Y is always in $Y[1,1]$.

ANSWER: Consider $Y[i,j]$ for any i and j. By the property of a Young tableau, $Y[1,1] \leq Y[i,1] \leq Y[i,j]$. Therefore, $Y[1,1]$ is the minimum because the above is true for every i and j.

(c) (10 points) Give an algorithm to implement EXTRACT-MIN on a nonempty $m \times n$ Young tableau that runs in $O(m+n)$ time. Your algorithm should use a recursive subroutine that solves an $m \times n$ problem by recursively solving either an $(m-1) \times n$ or an $m \times (n-1)$ subproblem. Define $T(p)$, where $p = m + n$, to be the maximum running time of EXTRACT-MIN on any $m \times n$ Young tableau. Give and solve a recurrence for $T(p)$ that yields the $O(m+n)$ time bound.

ANSWER: Since the minimum element is always in $Y[1,1]$, EXTRACT-MIN can return the element stored in $Y[1,1]$ and set $Y[1,1] = \infty$ to indicate that the element does not exist anymore. However, this might put the Young tableau in an inconsistent state, namely, $Y[1,1] = \infty$ and Y is not empty. Here's the Young tableau above after extracting the minimum element:

∞	4	9	∞
3	8	16	∞
5	14	∞	∞
12	∞	∞	∞

To fix this, we will look at the following general problem: $Y[i,j] = \infty$ but $Y[i+1..m, j..n]$ is a Young tableau and $Y[i..m, j+1..n]$ is a Yong tableau. We recursively exchange $Y[i,j]$ and $\min(Y[i+1,j], Y[j+1,j])$ until we obtain a Young tableau. We can call this recursive procedure YOUNGIFY (analogous to HEAPIFY).

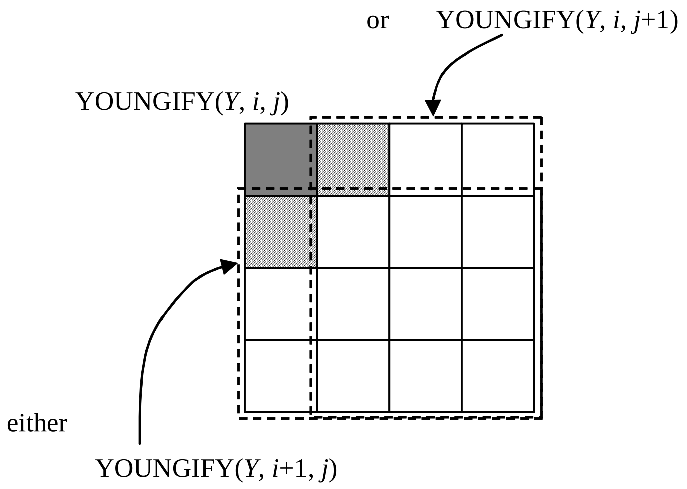
```
EXTRACT-MIN($Y$)
    $x \leftarrow Y[1,1]$
    $Y[1,1] \leftarrow \infty$
    YOUNGIFY($Y, 1, 1$)
    **return** $x$

YOUNGIFY($Y, i, j$)
    $smallest_i \leftarrow i$
    $smallest_j \leftarrow j$
    **if** $i+1 \leq m$ and $Y[i,j] > Y[i+1,j]$
      **then** $smallest_i \leftarrow i+1$
           $smallest_j \leftarrow j$
    **if** $j+1 \leq n$ and $Y[smallest_i, smallest_j] > Y[i, j+1]$
      **then** $smallest_i \leftarrow i$
           $smallest_j \leftarrow j+1$
    **if** $smallest_i \neq i$ or $smallest_j \neq j$
      **then** exchange $Y[i,j] \leftrightarrow Y[smallest_i, smallest_j]$
           YOUNGIFY($Y, smallest_i, smallest_j$)
```

Obviously, the running time of YOUNGIFY is $O(m+n)$ because YOUNGIY exchanges $Y[i,j]$ with either $Y[i+1,j]$ or $Y[i,j+1]$ and recursively YONGIFies the tableau at the corresponding entry. Therefore, with every recursive step, $i+j$ is incremented by 1. However, $i+j$ cannot be more than $m+n$ and this gives the time bound of $O(m+n)$. More precisely, we can say that YOUNGIFYing a tableau of size $m \times n$ will recursively YOUNGIFY a tableau of size $(m-1) \times n$ or of size $m \times (n-1)$.

Therefore, if we let $p = m+n$ to be the size of our problem:

$$T(p) = O(1) + T(p-1)$$

The solution of the above recurrence is $T(p) = O(p) = O(m+n)$.



(d) (10 points) Show how to insert a new element into a nonfull $m \times n$ Young tableau in $O(m+n)$ time.

ANSWER: If the Young tableau is not full, then $Y[m,n] = \infty$ (because otherwise, the tableau is full as argued in part (a)). Therefore, we can set $Y[m,n]$ to the new element that we want to insert. Although this will put the new element into the tableau, it might result in an inconsistent state, i.e. $Y[m,n]$ is not large enough to satisfy the property of a Young tableau. We can solve this problem by recursively retracting the new element and exchanging it with either the left or the top element (which ever is larger) until we obtain a Young tableau. We can call this procedure DECREASE-KEY.

```
INSERT($Y, k$)
    DECREASE-KEY($Y, m, n, k$)

DECREASE-KEY($Y, i, j, k$)
    **if** $Y[i,j] \leq k$
        **then return** error
    $Y[i,j] \leftarrow k$
    $threshold \leftarrow \infty$
    $largest_i \leftarrow i$
    $largest_j \leftarrow j$
    **while**($i > 1$ or $j > 1$) and $Y[i,j] < threshold$
        **do** exchange$Y[i,j] \leftrightarrow Y[largest_i, largest_j]$
            $i \leftarrow largest_i$
            $j \leftarrow largest_j$
            **if** $i-1 \geq 1$ and $Y[i,j] < Y[i-1,j]$
                **then** $largest_i \leftarrow i-1$
                     $largest_j \leftarrow j$
            **if** $j-1 \geq 1$ and $Y[largest_i, largest_j] < Y[i,j-1]$
                **then** $largest_i \leftarrow i$
                     $largest_j \leftarrow j-1$
            $threshold \leftarrow Y[largest_i, largest_j]$
```

Obviously, the running time of DECREASE-KEY is $O(m+n)$ because the element cannot retract more than $m+n$ times.

We can also use a recursive procedure similar to YOUNGIFY that works backward. Let's call it YOUNGIFY'.

```
INSERT($Y, k$)
    $Y[m, n] \leftarrow k$
    YOUNGIFY'($Y, m, n$)

YOUNGIFY'($Y, i, j$)
    $largest_i \leftarrow i$
    $largest_j \leftarrow j$
    if $i - 1 \geq 1$ and $Y[i, j] < Y[i-1, j]$
        then $largest_i \leftarrow i - 1$
             $largest_j \leftarrow j$
    if $j - 1 \geq 1$ and $Y[largest_i, largest_j] < Y[i, j-1]$
        then $largest_i \leftarrow i$
             $largest_j \leftarrow j - 1$
    if $largest_i \neq i$ or $largest_j \neq j$
        then exchange $Y[i, j] \leftrightarrow Y[largest_i, largest_j]$
             YOUNGIFY($Y, largest_i, largest_j$)
```

(d) (10 points) Using no other sorting method as subroutine, show how to use an $n \times n$ Young tableau to sort n^2 numbers in $O(n^3)$ time. Based on this, describe a sorting algorithm that has an $O(n^{1.5})$ worst-case running time.

ANSWER: Given n^2 numbers, we can insert them into an $n \times n$ Young tableau using INSERT. The sequence of n^2 inserts will take $n^2.O(n + n) = O(n^3)$ time. After that, we can repeatedly perform an EXTRACT-MIN n^2 times to obtain the numbers in order. This will take $n^2.O(n + n) = O(n^3)$ time also.

Assume A has length $m = n^2$ and Y is an empty $n \times n$ Young tableau.

```
SORT($A$)
    for $i \leftarrow 1$ to $m$
        do INSERT($Y, A[i]$)
    for $i \leftarrow 1$ to $m$
        do $A[i] \leftarrow$ EXTRACT-MIN($Y$)
```

In general, given n numbers to sort, let $a = \lceil \sqrt{n} \rceil$. We can create an empty $a \times a$ Young tableau and perform the above algorithm (with $m = n$). Each operation on the Young tableau will take $O(a + a) = O(\sqrt{n})$ time. Therefore, the running time of the algorithm will be $O(n\sqrt{n}) = O(n^{1.5})$.

(e) (0 points) I will not ask: given n distinct elements, how many Young tableaus can you make?

Problem 2: Sometimes you just have to count
(10 points) Describe an algorithm that, given n integers in the rangle 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a rangle $[a..b]$ in $O(1)$ time. Your algorithm should use $\Theta(n+k)$ prepreocessing time. For full credit you should

- Describe your algorithm in english
- Provide a pseudocode
- Provide arguments for the correctness of your algorithm
- Provide arguments for the running times of the preprocessing and the query operations

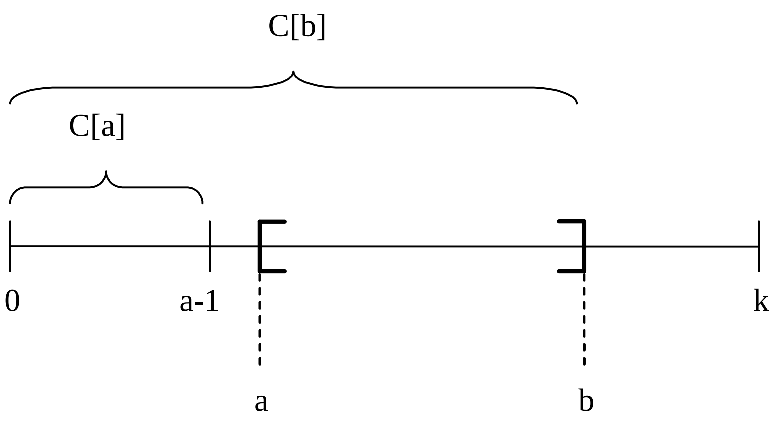
ANSWER: The algorithm will use a counter array $C[0..k]$ and set $C[i]$ to be the number of elements less or equal to i. This is identical to the first part of Counting sort and will take $\Theta(n+k)$ time. After that, to obtain the number of elements in the range $[a..b]$, it is enough to compute $C[b] - C[a-1]$ which can be done in $O(1)$ time.

pre-processing

```
for $i \leftarrow 0$ to $k$
    do $C[i] \leftarrow 0$
for $i \leftarrow 1$ to $n$
    do $C[A[i]] \leftarrow C[A[i]] + 1$
for $i \leftarrow 1$ to $k$
    do $C[i] \leftarrow C[i-1] + C[i]$
```

query

```
if $a = 0$
  then return $C[b]$
  else return $C[b] - C[a-1]$
```



Problem 3: S0rt1ing revisited
Suppose that we have an array of n data records to sort and that the key of each record has the value 0 or 1. An algorithm for sorting such a set of records might possess some subset of the following three desirable properties:

1. The algorithm runs in $O(n)$ time
2. The algorithm is stable
3. The algorithm sorts in place, using no more than a constant amount of storate space in addition to the original array

For full credit you should in each case:

- Describe your algorithm in english
- Provide a pseudocode, unless you are describing an algorithm for which we have seen the pseudocode in class
- Provide arguments that your algorithm possesses the desired properties

(a) (5 points) Describe an algorithm that possesses properties (1) and (2).

ANSWER: Counting sort runs in $O(n)$ time and is a stable sorting algorithm. No need to say more, as seen in class.

(b) (5 points) Describe an algorithm that possesses properties (2) and (3).

ANSWER: Insertion sort is an inplace sorting algorithm. It is also stable. To see this consider $A[i] = A[j]$ and $i < j$. Since $i < j$, $A[i]$ will be considered first. $A[i]$ will be added at the correct position (by shifting) into the sorted array $A[1..i-1]$. This will result in a sorted array $A[1..i]$ containing the original $A[i]$ at some position $k \leq i$. So $A[i]$ is now $A[k]$. When $A[j]$ is considered, $A[j]$ has to be shifted down into $A[1..j-1]$ of which $A[1..i]$ is a subarray containing $A[k]$ (originally $A[i]$). $A[j]$ cannot bypass $A[k]$ in the shifting process because $A[k] = A[j]$. Therefore, the original $A[i]$ and the original $A[j]$ will preserve their relative order.

(c) (10 points) Describe an algorithm that possesses properties (1) and (3).

ANSWER: We can perform one pass of a Quicksort PARTITION algorithm around the pivot $x = 0$. If it is a Lomuto PARTITION, it will place all elements ≤ 0 (all 0's) on the left and all elements > 0 (all 1's) on the right. Effectively, this sorts the array. It is inplace, and it has a $\Theta(n)$ running time. The only thing is that we have to work with a fictitious pivot that is not necessarily an element of the array, and hence we do not have to worry about correctly placing that pivot.

```
$i \leftarrow 0$
for $j \leftarrow 1$ to $n$
    do if $key[A[j]] \leq 0$
        then $i \leftarrow i + 1$
            exchange $A[i] \leftrightarrow A[j]$
```