# Sample Solution for Problem Set 3

Data Structures and Algorithms, Fall 2019

October 15, 2019

**6.1-7 (CLRS 3ed).** We first argue the node indexed by $\lfloor n/2 \rfloor + 1$ has no children. In particular, if the node has a left child, then the index of its left child will be:

$$2(\lfloor n/2 \rfloor + 1) > 2(n/2 - 1) + 2 \cdot 1 = n$$

Since the index of the left child is larger than the size of the heap, the node does not have children and thus is a leaf. The same argument goes for all nodes with larger indices.

On the other hand, for the node indexed by $\lfloor n/2 \rfloor$, we claim it is not a leaf: in case $n$ is even, it will have left child with index $n$; and in case $n$ is odd, it will have a left child with index $n - 1$ and a right child with index $n$.

**6.2-5 (CLRS 3ed).** See Figure 1 for the pseudocode.

```
MAX-HEAPIFY($A, i$)
while ($true$) do
    $left$ = LEFT($i$)
    $right$ = RIGHT($i$)
    if ($left < A.heap\_size$ and $A.nodes[left] > A.nodes[i]$) then
        $largest = left$
    else
        $largest = i$
    if $right < A.heap\_size$ and $A.nodes[right] > A.nodes[largest]$ then
        $largest = right$
    if ($largest == i$) then
        return
    SWAP($A.nodes[i], A.nodes[largest]$)
    $i = largest$
```

**Figure 1**

**6.3-3 (CLRS 3ed).** Let $P(h)$ be the assertion that "the number of nodes of height $h \leq \lceil n/2^{h+1} \rceil$ in any $n$-element heap". We will prove it via induction on $h$.

First we check the basis: $P(0)$ is the statement that "the number of leaves is $\lceil n/2 \rceil$ in any $n$-element heap", and this is certainly true (recall Exercise 6.1-7).

Now we verify the inductive step. Let $h \in \mathbb{Z}^+$ and suppose $P(h-1)$ holds. We wish to show that $P(h)$ holds.

Let's take a heap (which is also a tree) and remove all its leaves. We get a new heap with $n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$ elements. Note that the nodes with height $h$ in the old heap have height $h - 1$ in the new one.

We will calculate the number of such nodes in the new heap. By the induction hypothesis we know the number of nodes with height $h-1$ in the new heap is:

$$\left\lceil \lfloor n/2 \rfloor / 2^{(h-1)+1} \right\rceil < \left\lceil (n/2)/2^{h} \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

As mentioned, this is also the number of nodes with height $h$ in the original heap.

By mathematical induction we conclude that the assertion holds for all $n$-element heap.

**6.4-3 (CLRS 3ed).** Exercise 6.4-5 essentially implies the right answer for this exercise: the time complexity is $\Theta(n \lg n)$ in both cases. We were hoping to come up with solutions that are correct and easier to understand (compared with the one for Exercise 6.4-5), but have not got any luck yet. (In fact, we do have an alternative method for solving the decreasing order case, but again it is pretty complicated.) So, if you believe you have simple and *correct* method for solving this exercise, please let us know.

**6.4-5 (CLRS 3ed).** First consider the case in which $n = 2^h - 1$ for some $h$. In such scenario, the heap is initially a perfect binary tree.

Focus on the largest $(n+1)/2 = 2^{h-1}$ nodes in the heap. Verify that at least $2^{h-2}$ of these nodes are not leaves. (Try to prove this if you are not convinced.) Denote the set of these nodes by $S$. For these nodes in $S$, they have to be moved up to the root before they can be removed from the heap. So the running time of heap sort is lower bounded by the sum of the depths of these nodes.

Also note that at most $2^{h-3} - 1$ nodes can have depth smaller than $h-3$, so in the set $S$, at least $2^{h-2} - (2^{h-3} - 1) = 2^{h-3} + 1$ nodes have depth at least $h-3$. Thus the running time is lower bounded by $(2^{h-3}+1)(h-3) = \Omega(n \log n)$.

For the general case in which $n \neq 2^h - 1$ for any $h$, we can find the largest $h$ such that $n > 2^h - 1$. Since $2^h - 1 \geq n/2$, we know the running time $T_n \geq T_{n/2} = \Omega(n \log n)$.

**6.5-8 (CLRS 3ed).** See Figure 2 for the pseudocode.

```
HEAP-DELETE($A, i$)
1: HEAP-INCREASE-KEY($A, i, \infty$)
2: $A[1] \leftarrow A[A.heap\_size]$
3: $A.heap\_size \leftarrow A.heap\_size - 1$
4: MAX-HEAPIFY($A, 1$)
```

**Figure 2**

Proof of correctness: In the first step, node $i$'s value is changed to $\infty$. As a result, node $i$ will be moved to the root, and this process maintains heap property. The remaining lines simply remove the root from the heap. Again, this remove process maintains heap property.

**6-3 (CLRS 3ed).** **(a)** See the following table.

| | | | |
|---|---|---|---|
| 2 | 8 | 16 | $\infty$ |
| 3 | 9 | $\infty$ | $\infty$ |
| 4 | 12 | $\infty$ | $\infty$ |
| 5 | 14 | $\infty$ | $\infty$ |

**(b)** If $Y[1,1] = \infty$ for an $m \times n$ Young tableau $Y$, then for any $1 \leq i \leq m, 1 \leq j \leq n$, we have $Y[i,j] \geq Y[i-1,j] \geq Y[i-2,j] \geq \cdots \geq Y[1,j] \geq Y[1,j-1] \geq \cdots \geq Y[1,1] = \infty$, thus $Y[i,j] = \infty$. Since the choice of $i$ and $j$ are arbitrary, $Y[i,j] = \infty$ for all $i$ and $j$, implying $Y$ is empty.

On the other hand, if $Y[m,n] < \infty$ for an $m \times n$ Young tableau $Y$, then for any $1 \le i \le m, 1 \le j \le n$, we have $Y[i,j] < Y[i+1,j] < Y[i+2,j] < \cdots < Y[m,j] < Y[m,j+1] < \cdots < Y[m,n] < \infty$, thus $Y[i,j] < \infty$. Since the choice of $i$ and $j$ are arbitrary, $Y[i,j] < \infty$ for all $i$ and $j$, thus $Y$ is full.

**(c)** See figure 3 for the pseudocode.

```
Extract-Min
1:  procedure EXTRACT-MIN($Y, i, j$)
2:      if ($i = m$ and $j = n$) then
3:          $r \leftarrow Y[m,n]$
4:          $Y[m,n] \leftarrow \infty$
5:          return $r$
6:      else if ($i = m$ or ($j < n$ and $Y[i,j+1] < Y[i+1,j]$)) then
7:          SWAP($Y[i,j]$, $Y[i,j+1]$)
8:          return EXTRACT-MIN($Y, i, j+1$)
9:      else
10:         SWAP($Y[i,j]$, $Y[i+1,j]$)
11:         return EXTRACT-MIN($Y, i+1, j$)
```

**Figure 3**

It is easy to see that $Y[1,1]$ contains the minimum value. The code works by moving $Y[1,1]$ to $Y[m,n]$, while carefully maintaining the properties of the Young tableau. In particular, when $Y[1,1]$ has been moved to $Y[i,j]$, in the next step, we always exchange $Y[i,j]$ with the smaller of $Y[i+1,j]$ and $Y[i,j+1]$. As for the time complexity, clearly $T(p) = T(p-1) + O(1)$, thus $T(p) = O(p) = O(m+n)$.

**(d)** See figure 4 for the pseudocode.

```
Insert($Y, v$)
1: $i \leftarrow m, j \leftarrow n$
2: $Y[i,j] \leftarrow v, Y[0,\cdots] \leftarrow \langle -\infty, -\infty, \cdots, -\infty \rangle, Y[\cdots,0] \leftarrow \langle -\infty, -\infty, \cdots, -\infty \rangle$
3: while ($Y[i,j] < Y[i-1,j]$ or $Y[i,j] < Y[i,j-1]$) do   ▷ Recall $Y[i,j] = -\infty$ if $i = 0$ or $j = 0$.
4:     if ($Y[i-1,j] < Y[i,j-1]$) then
5:         SWAP($Y[i,j], Y[i,j-1]$)
6:         $j \leftarrow j-1$
7:     else
8:         SWAP($Y[i,j], Y[i-1,j]$)
9:         $i \leftarrow i-1$
```

**Figure 4**

The idea is essentially the same as in part **(c)**.

**(e)** We have a list to sort. First we insert the $n^2$ elements into an initially empty Young tableau one by one, then we extract the minimum element from the Young tableau for $n^2$ times. Each operation takes $O(n)$ time, and there are $O(n^2)$ operations, so the total runtime is $O(n^3)$.

**(f)** See figure 5 for the pseudocode.

The code works by eliminating one column or one row in each iteration. Specifically, we start at $Y[1,n]$. When we are at $Y[i,j]$, the code ensures for all $Y[i',j']$ such that $i' < i$ or $j' > j$, we have $Y[i',j'] \neq v$. Now, if $Y[i,j] > v$, then $Y[i,m] \ge Y[i,m-1] \ge \cdots \ge Y[i,j] > v$, so column $j$ is eliminated, and we move to $Y[i,j-1]$. Similarly, if $Y[i,j] < v$, we can move to $Y[i+1,j]$.

Since we only eliminate columns and rows that have no elements equal to $v$, if there is a cell in the Young tableau that equals to $v$, we will not eliminate it and will eventually find it.

**Additional Problem One.** We should not believe the professor. Suppose the best algorithm to multiply two $n-bit$ numbers (denoted as algorithm $A$) takes $T(n)$ time, and the best algorithm to square a $n-bit$

```
Find(Y, v)
    i ← 1
    j ← n
    while (True) do
        if (Y[i, j] = v) then
            return True
        if (Y[i, j] > v) then
            j ← j − 1
        else
            i ← i + 1
    return False
```

**Figure 5**

number (denoted as algorithm $B$) takes $F(n)$ time. Obviously $T(n) = \Omega(n)$ and $F(n) = \Omega(n)$, because reading a $n$-bit number already takes $\Omega(n)$ time.

The professor's view can be translated as $F(n) = o(T(n))$. Now consider the following algorithm $A'$, the goal of which is to multiply two $n$-bit numbers. After $A'$ gets input $a$ and $b$ (without loss of generality and for convenience, we assume $a \geq b$), it uses $O(n)$ time to determine $a \geq b$ and compute $c = a - b$, and then uses procedure $B$ to square $c$ and $a$ and $b$. The number of bits of $a$ and $b$ are $n$, so $c$ has at most $n$ bits as well. Thus, the squaring procedure costs $O(3 \cdot F(n)) = O(F(n))$ time. Then, $A'$ computes $d = a^2 + b^2 - (a - b)^2 = 2ab$, which costs $O(2n) = O(n)$ time (since the number of bis of $a^2$ and others are at most $2n$). We don't need to care about the case that the number may be negative since $a^2 + b^2 \geq (a - b)^2$. At last we use $O(n)$ time to compute $2ab/2$, which is right shift $d$ by one bit. At this point, we have obtained $ab$.

The running time of $A'$ is

$$T'(n) = O(n) + O(F(n)) + O(n) + O(n) = O(F(n)) + O(n) = O(F(n))$$

The last equal sign is because $F(n) = \Omega(n)$. Thus $T'(n) = o(T(n))$, which contradict our assumption that algorithm $A$ is the best algorithm for multiplication. So, we should not believe the professor.

**Additional Problem Two.** **(a)** Use $\phi(k)$ to denote the following proposition "If the input array $A$ to `Unusal` satisfies: the size of $A$ is $2^k$ and the subarray $A[1, ..., 2^{k-1}]$ and $A[2^{k-1} + 1, ..., 2^k]$ are both in increasing order, then `Unusal` correctly sorts the array (i.e., $A$ becomes in increasing order after executing `Unusal`)." Use $\psi(k)$ to denote the following proposition "On input array $A$ of size $2^k$, the procedure `Cruel` correctly sorts the array into increasing order."

We first use mathematical induction to prove $\phi(k)$ holds for $k \geq 1$. For $k = 1$, the assumption obviously holds due to lines $2, 3$ in `Unusal`. Now suppose $\phi(k)$ is true, we now prove $\phi(k + 1)$ is true. Denote the input array as $(a_1, a_2, ..., a_{2^{k+1}})$, and define $A_1, A_2, A_3, A_4$ as: $A_1 = (a_1, ..., a_{2^{k-1}}), A_2 = (a_{2^{k-1}+1}, ..., a_{2^k}), A_3 = (a_{2^k+1}, ..., a_{2^k+2^{k-1}}), A_4 = (a_{2^k+2^{k-1}+1}, ..., a_{2^{k+1}})$. For convenience we use $(A_1, A_2, A_3, A_4)$ to denote array $(a_1, ...., a_{2^{k+1}})$. Notice, $(A_1, A_2)$ is in increasing order, so is $(A_3, A_4)$. This further implies each of $A_1$, $A_2$, $A_3$, and $A_4$ is in increasing order. After the 6th line of the procedure, array $A$ becomes $(A_1, A_3, A_2, A_4)$. After the 8th line of the procedure, denote array $A$ as $(A'_1, A'_3, A'_2, A'_4)$. Using induction hypothesis, $(A'_1, A'_3)$ and $(A'_2, A'_4)$ are both in increasing order. Moreover, $(A_1, A_3)$ and $(A'_1, A'_3)$ contain same set of elements; similarly, $(A_2, A_4)$ and $(A'_2, A'_4)$ contain same set of elements. Now, for each $a \in A'_1$, if $a \in A_1$, then for every $b \in A_2, a \leq b$. Moreover, there must exist $c \in A'_3$ such that $c \in A_3$ ($A'_1$ can't contain all elements of $A_3$ as there already has an $a \in A'_1$ and $a \in A_1$). Thus, for every $d \in A_4, a \leq c \leq d$. So, at least $(3/4) \cdot 2^{k+1}$ elements are not less than $a$. (Particularly, $a$ is no larger than all elements in $A_2$, $A_4$, and the largest half in $(A_1, A_3)$.) Similar argument can be applied when $a \in A_3$. As a result, $A'_1$ contains the smallest $2^{k-1}$ elements in $(A_1, A_2, A_3, A_4)$. Similarly, we can deduce $A'_4$ contains the largest $2^{k-1}$ elements in $(A_1, A_2, A_3, A_4)$.

Lastly, after line 9 of the procedure, $(A'_3, A'_2)$ is sorted as $(A''_3, A''_2)$, and the array $(A'_1, A''_3, A''_2, A'_4)$ is in the right order. At this point, we have proved $\phi(k)$ holds for all $k \geq 1$.

Now we want to prove $\psi(k)$ holds for all $k \geq 1$. For $k = 1$, clearly the claim holds. Assume $\psi(k)$ is true, we now prove $\psi(k+1)$ is also true. Denote the input array as $(a_1, ..., a_{2^{k+1}})$. By induction hypothesis, before calling `Unusal`, $(a_1, ..., a_{2^k})$ and $(a_{2^k+1}, ..., a_{2^{k+1}})$ have been sorted. Since $\phi(k+1)$ holds, `Unual` rearranges $A$ into sorted order, implying $\psi(k+1)$ is true. This completes the proof.

**(b)** It is easy to verify on input $(3, 4, 1, 2)$ the algorithm outputs $(3, 1, 4, 2)$, which is wrong.

**(c)** It is easy to verify on input $(3, 4, 1, 2)$ the algorithm outputs $(3, 1, 2, 4)$, which is wrong.

**(d)** Denote the time complexity of `Curel` and `Unual` as $T(n)$ and $F(n)$ respectively, then we have recurrences:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + F(n), F(n) = 3 \cdot F\left(\frac{n}{2}\right) + O(n)$$

Using the master theorem, we get:

$$F(n) = O(n^{\lg 3}), T(n) = O(n^{\lg 3})$$