# Answer to Module 7 Problem Set

- Student Name: Gaole Dai
- Student ID: S01435587

## Problem 1

```python
def hp_build_heap(arr):
    heap = []
    for v in arr:
        hp_ins(heap, v)
    return heap
```

The complexity is calculated as

$$T = \mathcal{O}(\log(1)) + \mathcal{O}(\log(2)) + \ldots + \mathcal{O}(\log(n))$$

$$= \mathcal{O}(\log(1 * 2 * 3 * \ldots * n)) = \mathcal{O}(\log(n!))$$

The upper bound of every insertion `hp_ins` step is $\mathcal{O}(\log(n))$, since $n! \le n^n$, $\log(n!) \le \log(n^n) = n\log(n)$. Also, on the other hand, there is $n$ operations in total, the upper bound for building heap is the sum of all those operations, which is $\mathcal{O}(\log(n)) + \mathcal{O}(\log(n)) + \ldots + \mathcal{O}(\log(n)) = \mathcal{O}(n\log(n))$.

## Problem 2

**[12, 11, 4, 9, 10, 1, 3, 0, 2, 7]**

The Python code for standard build_heap algorithm is shown below.

```python
def heapify(arr, n, i):
    parent = int(((i-1)/2))
    if arr[parent] > 0:
        if arr[i] > arr[parent]:
            arr[i], arr[parent] = arr[parent], arr[i]
            heapify(arr, n, parent)

def insertNode(arr, key):
    n = len(arr)
    n += 1
    arr.append(key)
    heapify(arr, n, n-1)

def printArr(arr, n):
    for i in range(n):
        print(arr[i], end=" ")
    print()

inputArr = [4, 2, 7, 9, 12, 1, 3, 0, 10, 11]
n = 10
res = []
for i in range(n):
    key = inputArr[i]
    insertNode(res, key)
printArr(res, n)
```

The `heapify` middle outputs are:

4
4 2
7 2 4
9 7 4 2
12 9 4 2 7
12 9 4 2 7 1
12 9 4 2 7 1 3
12 9 4 2 7 1 3 0
12 10 4 9 7 1 3 0 2
12 11 4 9 10 1 3 0 2 7

## Problem 3

**[12, 11, 7, 10, 4, 1, 3, 0, 9, 2]**

The Python code for Floyd build_heap algorithm is shown below.

```
def heapify(arr, N, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2

    if l < N and arr[l] > arr[largest]:
        largest = l

    if r < N and arr[r] > arr[largest]:
        largest = r

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, N, largest)

def buildHeap(arr, N):

    startIdx = N // 2 - 1

    for i in range(startIdx, -1, -1):
        heapify(arr, N, i)

def printHeap(arr, N):
    for i in range(N):
        print(arr[i], end=" ")
    print()


arr = [4, 2, 7, 9, 12, 1, 3, 0, 10, 11]
N = len(arr)
buildHeap(arr, N)
printHeap(arr, N)
```

The `heapify` middle outputs are:

4 2 7 9 12 1 3 0 10 11

4 2 7 10 12 1 3 0 9 11

4 2 7 10 12 1 3 0 9 11

4 12 7 10 11 1 3 0 9 2

12 11 7 10 4 1 3 0 9 2

12 11 7 10 4 1 3 0 9 2

## Problem 4

**[11, 10, 4, 9, 7, 1, 3, 0, 2]**

The Python code for delMax algorithm is shown below.

The idea is to replace the root with the last element, then heapify the root.

```python
def swap(i, j, H) :
    temp = H[i]
    H[i] = H[j]
    H[j] = temp

def shiftDown(i, H, size) :
    maxIndex = i
    l = (2 * i) + 1
    if (l <= size and H[l] > H[maxIndex]) :
        maxIndex = l
    r = (2 * i) + 2
    if (r <= size and H[r] > H[maxIndex]) :
        maxIndex = r
    if (i != maxIndex) :
        swap(i, maxIndex, H)
        shiftDown(maxIndex, H, size)
def extractMax(size, H) :
    result = H[0]
    H[0] = H[size]
    size = size - 1
    shiftDown(0, H, size)
    return result

arr = [12, 11, 4, 9, 10, 1, 3, 0, 2, 7]
size = 10
extractMax(size-1, arr)
j = 0
while (j < size-1) :
    print(arr[j], end = " ")
    j += 1
print()
```

## Problem 5

The Python real code is shown below.

```
def heapify(arr, N, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2

    if l < N and arr[largest] < arr[l]:
        largest = l

    if r < N and arr[largest] < arr[r]:
        largest = r

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, N, largest)

def heapSort(arr):
    N = len(arr)

    # Build a maxheap.
    for i in range(N//2 - 1, -1, -1):
        heapify(arr, N, i)

    # Extract elements
    for i in range(N-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```

For pseudocode, it uses Floyd `heapify`, and use `buildHeap` first, then sort using the array-implemented heap.

```
def heapify(arr):
     for k in range(len(arr)//2 - 1,-1,-1):
         sink(arr,k)
     return arr
def heapSort(arr):
    # Build maxHeap
    for i in range(len(arr)//2 - 1, -1, -1):
        heapify(arr)
    # Extract the maximum element
    for i in range(len(arr)- 1, 0, -1):
        swap(arr[0], arr[i])
        heapify(arr)
```

## Problem 6

The complexity (worst case) is $\mathcal{O}(n\log(n))$.

According to the code implementation in Problem 5, `heapify` function is top-down method to go through the tree, the height of a binary tree with size $n$ is $\log_2(n)$ in maximum, thus the complexity of this function is $\mathcal{O}(\log(n))$.

Similarly, for building heap it calls `heapify` multiple times, the complexity is maximum $\mathcal{O}(n\log(n))$

For `heapSort` function, it calls the `heapify` function $(n-1)$ times, the cost is $(n-1)\log(n)$ which also indicate the complexity is $\mathcal{O}(n\log(n))$.

## Problem 7

The pseudocode for L-median algorithm is shown below.

```
# max heap for storing the small values, min heap for storing the max values.

maxHeap, minHeap = [], []

def insert(val):
    # Choose a heap to insert the value, heap_push is the push method for
priority queue
    if (len(maxHeap) == 0):
        heap_push(maxHeap, val)

    if (val > maxHeap[0]):
        heap_push(minHeap, val)
    else:
        heap_push(maxHeap, val)

    # Rebalance the heaps if unbalanced
    if len(maxHeap) > len(minHeap) + 1:
        tmp = heap_pop(maxHeap)
        heap_push(minHeap, tmp)
    else if len(maxHeap) + 1 < len(minHeap):
        tmp = heap_pop(minHeap)
        heap_push(maxHeap, tmp)

def show_lmedian():
    if (len(maxHeap) > len(minHeap)):
        return maxHeap[0]
    else:
        return minHeap[0]
```

The complexity for `heap_push` in `insert` would be $\mathcal{O}(\log(n))$, for heap rebalance step it is $\mathcal{O}(1)$ , since the element would be place at the top, no need to `heapify` to maintain the priority.

Apparently, the complexity for `show_lmedian` is $\mathcal{O}(1)$.