Yong, Zhou
HW_2

(1), Never true.

F(n) = o(g(n)) and f(n) = Ω(g(n))

- Prove: F(n) = o(g(n)) is strict upper bound, and f(n) = Ω(g(n)) is the lower bound. By their definition that 0 <= F(n) < cg(n) for every constant c > 0, there is a constant n_0>0 for all n > n_0 . However, f(n) = Ω(g(n)), for f(n) exist, c > 0, n_0 > 0, that 0 <= cg(n) <= f(n), for all n > n_0 . For F(n) = o(g(n)) and f(n) = Ω(g(n)), that f(n) < c*g(n) <= g(n) which cannot be.
  * Example:
    Let f(n) = n and g(n) = n^2 ; $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ = 0 and $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ = $\frac{1}{2}$
  Therefore, f(n) = o(g(n)) but log(f(n)) ≠ o(log(g(n))).

(2), sometime true

- F(n) = O(g(n)), $2^{f(n)} = 2^{O(g(n))}$, $2^{O(g(n))}$ = O($2^{g(n)}$), so O(g(n)) ≠ g(n); Example: n^2 + 5n = O(n^2), no other way around base on the definition of O-notation.
  - Let f(n) = 2 log n and g(n) = log n, we have $2^{f(n)} = n^2$ and $2^{g(n)}$ = n, Therefore, f(n) = O(g(n)) but $2^{f(n)}$ ≠ O($2^{g(n)}$)
- True example:
  - Let f(n) = n = g(n); $2^{f(n)} = 2^n = 2^{O(n)}$.

(3), sometime True

- f(n) = O(g(n)), g(n) – O(g(n)) = Ω(g(n)),
  - Correct example, g(n) = n^2 + 2n, f(n) = n^2 ; g(n) – f(n) = n^2 + 2n - n^2 = 2n, which is Ω(g(n));
  - False example, g(n) = n^2, f(n) = n^2 -1. g(n) – f(n) = n^2 - n^2 + 1 = 1, which is g(n) = f(n), is not Ω(n^2)

(4), True.

- f(n) = o(g(n)), g(n) – f(n) = Ω(g(n))
  - F(n) = o(g(n)) is strict upper bound, for every constant c > 0, there is a constant n_0 > 0 such that 0 <= f(n) < cg(n) for all n >= n_0.
  - For Ω(g(n)), which is the lower bound, to f(n): there exist constants w > 0, n_0 > 0, such that 0 <= wg(n) < f(n) for all n >= y_0.
  - g(n) – f(n) < c g(n). for all c>0, there is a n_0 > n. 0 <= f(n) < cg(n) for n >= n_0; for $\lim_{n\to\infty} \frac{g(n)}{f(n)} = g(n)$, $\lim_{n\to\infty} g(n)$ – f(n) = g(n) = Ω(g(n))

2,

Algorithm:

Initialize: K[i]: value at node(element) i

- Max-heap(K):
  K.heapsice = K.length
  For i in K.length/2 dowmto 1
  Max-heapify(K, i)

- Max-heapify(K, i):
  If left child <= K.heapsize and K[left Chile] > K[i]
  Largest = left Chile
  Else:
  Largest = i
  If right child <= K.heapsize and K[right Child] > K[largest]
  Largest = r
  If largest != i:
  Exchange K[i] with K[largest]
  Max-heapify (K, largest)

Correctness:

The pointer point to the min-heap size n, by definition, the pointer will pick the root node(the smallest value), then left node and right node(the smaller values than root node), and so on.

1). The max-heap, the size of input is the size of min-heap in the first k elements. Say array M.
2). The max-heap only use the first k elements of the min-heap, which is the smallest k elements.
3). Take the left children and right children of Kth element, compare them one by one with root of the max-heap.
4), Due to the root of max heap contains the largest value it means that compare the remaining elements of array with root to check if some smaller element in array exists which is not a part of max-heap.
4). If the element i in array M, M[i] less than the root of the max-heap, change the root element with M[i], to get smallest k elements.
5). Arrange one element in the max-heap take complexity of log(k) (complexity prove in the textbook p,64)
6). After completely traversing the part of input of min-heap (k smallest elements and it children), we got the smallest k elements, which in the max-heap. The k elements which take complexity of klog(k).