**CS 263**
**Data Structures**
# ASSIGNMENT # 1

1. Prove or disprove each of the following conjectures.

   a. $f(n) = O(g(n))$ implies $g(n) = O(f(n))$.
   **Solution:** This conjecture is false. We disprove by a counterexample. Let $f(n) = n$ and let $g(n) = n^2$. Then $f(n) = O(g(n))$ which can be seen by letting $c = 1$ and $n_0 = 1$. But $g(n) \neq O(f(n))$ which can be shown as follows. Assume for sake of contradiction that $g(n) = O(f(n))$. Then there is some $c > 0$ and $n_0 \geq 0$ such that $n^2 \leq cn$ for all $n \geq n_0$. But this is true if and only if $n \leq c$ for all $n \geq n_0$. But this is not true since for any choice of $n_0$ and $c$, we can pick $n = n_0 + c + 1$ (for example). Clearly $n$ is at least as large as $n_0$, but $n$ is greater than $c$.

   b. $f(n) = O((f(n))^2)$
   **Solution:** This conjecture is false. Again we will disprove by a counterexample. Let $f(n) = 1/n$. Then $f(n)^2 = 1/n^2$. Now assume that $f(n) = O(f(n)^2)$. Then there exists $c > 0$ and $n_0 \geq 0$ such that $1/n \leq c/n^2$. But this is equivalent to $n \leq c$. Again by the above argument this is a contradiction since $n$ is unbounded.

   c. $\sum_{x=1}^{n} \frac{x}{2^x} = O(1)$.
   **Solution:** The terms in the sum are all positive. Therefore the sum is less than $\sum_{x=1}^{\inf} \frac{x}{2^x}$. The only thing we need to show is the convergence of this series which follows from basic calculus. I.e., we want to show that $lim_{n\to\inf} \sum_{x=1}^{n} \frac{x}{2^x}$ converges. Use the ratio test:

$$\frac{a_{n+1}}{a_n} = \frac{(n+1)2^n}{2^{n+1}n} = \frac{n+1}{2n} = 1/2 + 1/n$$

   Thus

$$lim_{n\to\inf} \frac{a_{n+1}}{a_n} = lim_{n\to\inf}(1/2 + 1/n) = 1/2 < 1.$$

   Thus it converges to some constant, say $k$. Pick $c = k+1$ and $n_0 = 1$. Then we have $\sum_{x=1}^{n} \frac{x}{2^x} \leq c$ for all $n \geq n_0$.

3. Problem 6-2 from the book (edition 3).

   **Solution:**

   a. We will represent the heap in an array $A[1, \ldots, n]$. Root is at 1. The children for node $i$ will be $d(i-1)+1, \ldots, d(i-1)+d$. $Parent(i) = \lceil \frac{i-1}{d} \rceil$. $Child(i,j) = (i-1)d + j + 1$.

   b. Assuming a d-ary tree with only root node has height 0 (a.k.a. edge counting), the maximum number of nodes in a $d$-ary tree of height $h$ is $1 + d + d^2 + \ldots + d^h = \frac{d^{h+1}-1}{d-1}$. If the n elements complete the last layer of the d-ary tree exactly then it is an equality. Otherwise it is less than that. This gives an inclusive upper bound. The lower bound is a complete d-ary tree with height of $h-1$ and is exclusive. We have

$$\frac{d^h - 1}{d-1} < n \leq \frac{d^{h+1}-1}{d-1}$$

   from which it follows that

$$d^h < n(d-1) + 1 \leq d^{h+1}$$

   and taking the $lg_d$:

$$h < lg_d(n(d-1)+1) \leq h+1$$

h and h + 1 are consecutive integers, therefore applying the ceil() always yields an equality from the right inequality:

$$h = \lceil lg_d(n(d-1)+1) \rceil - 1$$

Expressed in Big-Oh notation:

$$h = \lceil \frac{lg(n(d-1)+1}{lgd} \rceil - 1 = O(\frac{lgdn}{lgd}) = O(\frac{lgn}{lgd}).$$

c.
```
function ExtractMax(A)
        maxElement = A[1]                                   ▷ index starts at 1
        exchange A[1] with A[A.heap-size]
        A.heap-size = A.heap-size - 1
        A[A.heap-size + 1] = null                ▷ deletes and prevents loitering
        SINK(A, 1)
        return maxElement
end function

function sink(A, k)
    while child(k,1) ≤ A.heap − size do   ▷ child(k,1) is the index of the first child of node k
        maxIndex = child(k,1)
        for i = 2 to d do
            if child(k, i) >A.heap-size then              ▷ checks for array out of bounds
                BREAK
            end if
            if A[child(k, i)] >A[maxIndex] then ▷ find the index of the children with max value
                maxIndex = child(k, i)
            end if
        end for
        if A[k] ≤ A[maxIndex] then                             ▷ No need to sink anymore
            BREAK
        end if
        exchange A[k] with A[maxIndex]
        k = maxIndex                                              ▷ point to new index
    end while
end function
```

In ExtractMax, all lines constant time except for the $sink(A, K)$ call, which has 2 nested loops. The outer loop takes time proportional to heihg of the heap, whcih we knwo from part b) is $O(\frac{lgn}{lgd})$. The inner loop takes time proportional to $O(d)$. Therefore the total running time is $O(dh) = O(\frac{dlgn}{lgd})$.

d.
```
function insert(A, newElement)
    A.heap-size = A.heap-size + 1                       ▷ increase size of heap by 1
    A[A.heap-size] = newElement                     ▷ put new element at end of heap
    swim(A, A.heap-size)          ▷ floats element up d-ary tree to maintain the heap property
end function

function swim(A, k)
    while k >0 AND A[parent(k)] >A[k] do
        exchange A[k] with A[parent(k)]
        k = parent(k)
    end while
```

**end function**

In Insert, all lines take constant time except for the $swim(A, k)$ call, which has running time proportional to height of the heap, which we know from part b) is $O(\frac{lgn}{lgd})$.

e. **function** IncreaseKey(A, i, k)
   **if** k <A[i] **then**
      throw error
   **end if**
   A[i] = k
   swim(A,i)
**end function**

The algorithm is similar to the binary heap algorithm and the running time is proportional to the height, $O(\frac{lgn}{lgd})$.

4. Give an algorithm that uses one of the data structures that we have studied so far to perform the following. The input consists of $k$ sorted lists $L_1, \ldots, L_k$, each one containing a list of $n/k$ integers in increasing order. The algorithm should output a single list $L$ that contains the $n$ integers in $A_1, \ldots, A_k$, sorted in increasing order.

   a. Give a simple algorithm for solving the above problem with worst-case time complexity $O(n \log k)$. Explain why it works, and why it has worst-case time complexity $O(n \log k)$. *Give a clear and consise description of your algorithm in English. Do not use pseudocode.*

   **Solution:** The basic idea is to maintain a MinHeap that contains $k$ elements, specifically the smallest integer from each one of the $k$ sorted lists $A_1, \ldots, A_k$. More precisely:

   1) First build a Min Heap that contains the following $k$ elements: $(a_1, 1), (a_2, 2), \ldots, (a_j, j), \ldots, (a_k, k)$ where each $a_j$ is the smallest element in the sorted list $A_j$, and the $a_j$'s are used as the heap keys. Remove each $a_j$ from $A_j$.
   2) Then repeatedly do the following.
      a) First do an Extract-Min to find and remove the element with the smallest key from the Min Heap; say this element is $(x, i)$. Output $x$.
      b) Note that the above $x$ came from list $A_i$. Remove the smallest (remaining) element from the list $A_i$, say it is $y$, and insert $(y, i)$ into the Min Heap. Note: if $A_i$ is empty, then skip the second step. In this case, the Min Heap size decreases by one because the element $(x, i)$ that was removed from the Min Heap in step (a) is not replaced.

   b. Explain why your algorithm's worst-case time complexity is $O(n \log k)$.

**Solution:**

1) The initial Min Heap contains $k$ keys. So it takes at most $O(k)$ time to build it using $BuildMinHeap$ (a procedure that is very similar to BuildMaxHeap of Section 6.3.) If we build it by doing $k$ repeated inserts, this takes at most $O(k \log k)$ time.
2) Each one of the $n$ outputs requires:
   a) one ExtractMin, which takes $O(\log k)$ time in the worst case and
   b) at most one Insert, which also takes $O(\log k)$ time in the worst case. So the worst case time complexity of the $n$ outputs is $O(k \log k)$.

So overall, the worst case time complexity of the above algorithm is $O(k) + O(n \log k)$ (or $O(k \log k) + O(n \log k)$ if we used $k$ repeated inserts to build the initial Min Heap). Since $k \leq n$, the worst-case time complexity is $O(n \log k)$.