CS:3330 Algorithms
Midterm Exam (100 points)

Closed books and notes (except one sheet of notes)
None of the digital devices is allowed.
3/20/2018

1. (25 points) List the following 7 functions according to their growth rate:

\[(n \log n)^8, \; 2^n, \; (n/2)^{10}, \; (n/4)^{n^{1/4}}, \; (\log n)^{\log(n)}, \; n^{\log(\log(n))}, \; (\log(\log n))^n\]

**Answer:** \[(n \log n)^8, \; (n/2)^{10}, \; (\log n)^{\log(n)} = n^{\log(\log(n))}, \; 2^n, \; (\log(\log n))^n, \; (n/4)^{n^{1/4}}\]

2. (25 points) We insert the following numbers in the given order into an empty AVL tree. For each rotation in an insertion, please display the tree before and after rotation. If no rotation happens during an insertion, simply draw the tree after the insertion:

9, 6, 1, 5, 4, 3.

**Answer:**

- Insert 9:
  - Before:
    - Insert 6:
      - Before:
        - Insert 1:
          - Before:
            - Rotate:

- Insert 5:
  - Before:
    - Insert 4:
      - Before:
        - Rotate:

- Insert 3:
  - Before:
    - Rotate:
3. (25 points) In-place heapsort is the only known in-place comparison-based sorting algorithm whose worst case time complexity is $O(n \log n)$. Suppose the array $A$ contains four numbers 1, 4, 9, 5, and the only operation to change the array $A$ is by $\text{swap}(i, j)$, which swaps the elements of $A$ at positions $i$ and $j$. Please illustrate how the heap is constructed and changed by every swap operation (show the content of the array $A$ after each swap operation).

**Answer:** The array $A = [3, 8, 2, 6]$ represents the binary tree:

The in-place heapsort uses the bottom up construction to construct a max-heap. We will use $\text{sw}(A[i], A[j])$ instead of $\text{swap}(i, j)$ for readability.

The first swap: $\text{sw}(8, 3)$

$A = [8, 3, 2, 6]$

The second swap: $\text{sw}(3, 6)$

$A = [8, 6, 2, 3]$

Now the heap is complete.

The second phase of the in-place heapsort is to repeatedly swap the maximal element with the last element of the heap and reduce the size of the heap by one.

The third swap: $\text{sw}(8, 3)$

$A = [3, 6, 2, 8]$

Repair the heap by

The fourth swap: $\text{sw}(3, 6)$

$A = [6, 3, 2, 8]$

The fifth swap: $\text{sw}(6, 2)$

$A = [2, 3, 6, 8]$

Repair the heap by

The sixth swap: $\text{sw}(2, 3)$

$A = [3, 2, 6, 8]$

The seventh swap: $\text{sw}(3, 2)$

$A = [2, 3, 6, 8]$

Now $A$ is sorted.
4. (25 points) The following algorithm will merge n sorted lists into a single sorted list:

```java
mergeAll(L, n) {
    r = L[0];
    for (j = 1; j < n; j++) r = merge(r, L[j])
    return r;
}
```

In the above pseudo code, merge(L₁, L₂) will merge L₁ and L₂ into a single sorted list and the complexity of merge(L₁, L₂) is \(O(|L₁| + |L₂|)\), where \(|X|\) denotes the length of list X. If L[j] is sorted and \(|L[j]| = m\) for all different j, please analyze the complexity of mergeAll in terms of n and m. If you think the algorithm mergeAll can be improved, please design an efficient algorithm (as fast as you can) which will do the same job and analyze the complexity of your algorithm in terms of n and m.

**Answer:** In the for loop of mergeAll, \(|r|\) grows from m, 2m, 3m, ..., to nm. The cost of merge(r, L[j]) will be 2m, 3m, ..., nm. Adding them together, we have

\[2m + 3m + ... + nm = O(mn^2).\]

A better algorithm is to merge sorted lists pairwise, not accumulate into one list.

```java
mergeAll2(L, n) {
    while (n > 1) {
        int k = n/2;  // k is the floor of n/2
        for (j = 0; j < k; j++) L[j] = merge(L[j], L[n-j-1]);
        if (n % 2 == 0) n = k;
        else n = k+1;
    }
    return L[0];
}
```

The complexity of this algorithm is \(O(mn \log(n))\), because in the for loop, each element in L is merged at most once, so the cost of the for loop is \(O(mn)\), because L contains mn elements. The while loop can go at most \(\log(n)\) times, because n is halved in each iteration.

Another algorithm is to use the exact structure of mergesort:

```java
mergeAll3(L, low, high) {
    if (low == high) return L[low];
    int mid = (low+high)/2;
    x = mergeAll3(L, low, mid);
    y = mergeAll3(l, mid+1, high);
    return merge(x, y);
}
```

The first call to mergeAll3 is mergeAll3(L, 0, n-1). The complexity is also \(O(mn \log(n))\). Another algorithm is to use a min-heap of n elements, each element is a sorted list L[i], and its key is the first number in L[i]. We call deleteMin mn times, each takes \(O(\log(n))\) time, so the total time is still \(O(mn \log(n))\).