COMP 3711 Design and Analysis of Algorithms
2015 Fall
Solutions to Assignment 1

1. (a) $A = \Omega(B)$;
   (b) $A = O(B), A = \Omega(B), A = \Theta(B)$;
   (c) $A = O(B)$;
   (d) $A = O(B), A = \Omega(B), A = \Theta(B)$;
   (e) $A = \Omega(B)$.

2. (a) $T(n) = O(\log n)$.
   (b) $T(n) = O(n^2)$.
   (c) $T(n) = O(\log n)$.
   (d) $T(n) = O(n^{\log 3})$.
   (e) $T(n) = O(\log \log n)$.
   Expanding out the recurrence, we have

$$T(n) = T(n^{1/2}) + 1 = T(n^{1/4}) + 2 = T(n^{1/8}) + 3 = \cdots = T(n^{1/2^x}) + x,$$

   where x is the smallest integer such that $n^{1/2^x} \leq 2$, or $1/2^x \leq \log_n 2$, $2^x \geq \log n$, so $x \geq \log \log n$. So we have $x = \lceil \log \log n \rceil$, and $T(n) = O(\log \log n)$.

3. (a) The running time of merge is linear on the input arrays. We will be running this on arrays of size:
   $n + n$, $2n + n$, ..., $(k-1)n + n$

   The total cost is

$$\begin{aligned} & \left(n \sum_{i=1}^{k-1} i \right) + (k-1)n \\ = \ & n \left(\frac{k(k-1)}{2} \right) + (k-1)n \\ = \ & n \frac{k^2 - k}{2} + (k-1) \\ = \ & O(nk^2). \end{aligned}$$

   (b) We use divide-and-conquer, in a way similar to merge sort. We first divide the k sorted arrays into two halves, recursively merge each half, and then merge the two halves together.

```
MULTI-MERGE(A[1..k][1..n], i, j):
if i = j then
  └ return A[i][1..n];
m ← ⌊(i+j)/2⌋;
return MERGE(MULTI-MERGE(A, i, m), MULTI-MERGE(A, m+1, j));
```

The initial call to this recursive algorithm is MULTI-MERGE$(A, 1, k)$.

Let $T(k)$ be the running time of the algorithm on k sorted lists. We have the recurrence $T(k) = 2T(k/2) + O(nk)$ and $T(1) = O(n)$, which solves to $T(k) = O(nk \log k)$.

4. If $n \leq 3$ we can solve the problem trivially. Let $m = \lfloor n/2 \rfloor$. We look at the three elements $A[m-1], A[m], A[m+1]$. There could be the following cases:

   (a) If $A[m-1] > A[m]$ and $A[m] < A[m+1]$, then $A[m]$ is a local minimum and we are done;

   (b) If $A[m-1] < A[m] < A[m+1]$, then by the boundary condition there must be at least one local minimum between $A[1]$ and $A[m]$, so we recursively solve the problem on $A[1..m]$;

   (c) If $A[m-1] > A[m] > A[m+1]$, similar to the case above, we recursively solve the problem on $A[m..n]$;

   (d) If $A[m-1] < A[m]$ and $A[m] > A[m+1]$, we can recurse into either $A[1..m]$ or $A[m..n]$, but not both.

   In any case, we either terminate or reduce the problem size by half. So we have the recurrence $T(n) \leq T(n/2) + O(1)$, which solves to $T(n) = O(\log n)$.

5. (a) We use another array $C[i]$ to remember whether i has been checked, and a variable m to remember how many indices have been checked.

```
RandomSearch(A, x):
n ← size of A;
m ← 0;
C[1..n] ← 0;
while m < n do
    j ← random(1, n);
    if C[j] = 0 then
        C[j] ← 1;
        m ← m + 1;
    else
        if A[j] = x then return A[j] ;
return nil;
```

   (b) This is the same as the waiting time problem where the success probability is $p = 1/n$. So the expected number of indices we pick until we find $A[i] = x$ is $1/p = n$.

   (c) This is the same as the waiting time problem where the success probability is $p = k/n$. So the expected number of indices we pick until we find $A[i] = x$ is $1/p = n/k$. Thus for larger k, the randomized algorithm is better than the deterministic algorithm.

   (d) This is the same as the coupon collector problem, so the expected number of indices is $O(n \log n)$. Note that this is worse than the deterministic algorithm.