Design and Analysis of Algorithms, Fall 2014
Exercise I: Solutions

I-1 Let $T(n) = M$ for $n \leq M$ for some constant $M \geq 1$ independent of n, and $2T(n) = 2T(n/2) + 3T(n/3) + n$ otherwise. Show that $T(n) = \Theta(n \log n)$.

As a clarification, we assume that the recurrence is defined for all positive reals. To show $T(n) = \Theta(n \log n)$, we show separately that $T(n) = O(n \log n)$ and $T(n) = \Omega(n \log n)$. All logarithms used are 2-based.

For the "O" direction, we show by induction that $T(n) \leq n \log n + cn = \Theta(n \log n)$ for $n \geq 1/3$ for some $c > 0$.

1. Case $1/3 \leq n \leq M$: It's easy to see that $n \log n$ is bounded from below by some constant c_0. Now, if we choose any $c \geq 3(M - c_0)$, we have that $T(n) = M \leq c_0 + (1/3)c \leq n \log n + cn$.
2. Induction step $n > M$: assume that the claim holds for $n/2$ and $n/3$. Note that, since $M \geq 1$, the smallest n for which the claim is assumed to hold is $1/3$, which is proven in the base case. Under these assumptions, we have that

$$\begin{array}{rcll} 2T(n) & = & 2T(n/2) + 3T(n/3) + n & \\ & \leq & 2(\frac{n}{2}\log\frac{n}{2} + \frac{n}{2}c) + 3(\frac{n}{3}\log\frac{n}{3} + \frac{n}{3}c) + n & \text{induction assumption} \\ & = & n(\log\frac{n}{2} + \log\frac{n}{3} + 1) + 2nc & \\ & = & n(2\log n - \log 6 + 1) + 2nc & \text{properties of logarithm} \\ & \leq & 2(n \log n + nc) & \log_2 6 > 1 \end{array}$$

   holds regardless of our choice of c.

For the "Ω" direction, we show by induction that $T(n) \geq cn \log n = \Theta(n \log n)$ for some $c > 0$.

1. Case $n \leq M$: by choosing $c \leq 1/\log M$, we have that $T(n) = M \geq cM \log M \geq cn \log n$.
2. Induction step $n > M$: assume that the claim holds for $n/2$ and $n/3$. Then,

$$\begin{array}{rcll} 2T(n) & = & 2T(n/2) + 3T(n/3) + n & \\ & \geq & 2(c\frac{n}{2}\log\frac{n}{2}) + 3(c\frac{n}{3}\log\frac{n}{3}) + n & \text{induction assumption} \\ & = & 2cn \log n + n(1 - c\log 6) & \text{properties of logarithm} \\ & \geq & 2cn \log n & \end{array}$$

   where the last inequality holds if $1 - c\log 6 \geq 0 \iff c \leq 1/\log 6$.

Since we want both of steps 1 and 2 to hold, we can choose $c = \min(1/\log M, 1/\log 6)$.

I-2 (**CLRS 2.3-7**) Describe a $\Theta(n \log n)$-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

Sort the set S using merge sort. Then for each $y \in S$ separately use binary search to check if integer $x - y$ exists in S. Sorting takes time $\Theta(n \log n)$. Binary search takes time $O(\log n)$ and is executed n times. The total time is thus $\Theta(n \log n)$.

If S is sorted, the problem can also be solved in linear time by scanning the list S at the same time forward and backward directions:

Input: list S sorted in ascending order, x
Output: true if there exist two elements in S whose sum is exactly x, false otherwise

```
1 i ← 1, j ← n
2 while i ≤ j do
3 |   if S[i] + S[j] = x then
4 |   |   return true
5 |   if S[i] + S[j] < x then
6 |   |   i ← i + 1
7 |   else
8 |   |   j ← j − 1
9 return false
```

Note: The above solutions assume that the two elements can actually be the same element (for examples, if $S = \{1,2,5\}$ and $x = 4$, the algorihms return true since $2+2=4$). If this is not allowed, then small changes to algorithms are needed. (In the first solution skip y if $2y = x$. In the second solution replace $\leq$ by $<$.)

I-3 (**CLRS 2-4 Inversions**) Let $A[1..n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an *inversion* of A.

a. List the five inversions of the array $(2,3,8,6,1)$.

(1,5), (2,5), (3,4) (3,5), (4,5)

b. What array with elements from the set $\{1,2,\ldots,n\}$ has the most inversions? How many does it have?

If the array is sorted in descending order, then every pair (i, j) with $i < j$ is an inversion. The number of such pairs is $n(n-1)/2$.

c. What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.

Assume a version of insertion sort that does not use a temporary variable but instead uses swaps to move the current element into the correct place in the already sorted part of the array. Now each execution of the inner loop body (that is, each swap) eliminates one inversion. Sorting eliminates all inversion and therefore the inner loop is executed exactly I times, where I is the number of inversions in the array. Since the outer loop is executed n times the total running time is $\Theta(I+n)$.

d. Give an algorithm that determines the number of inversions in any permutation of n elements in $\Theta(n \log n)$ worst-case time. (*Hint:* Modify merge sort.)

We modify merge sort to count the number of inversion while sorting the array. The number of inversion in array $A = BC$ is

$$I(A) = I(B) + I(C) + I(B,C),$$

where $I(X)$ is the number of inversion in array X and $I(X,Y)$ is the number of pairs $(x,y) \in X \times Y$ such that $x > y$. Let $head(X)$ be the first element in array X. We can compute term $I(B,C)$ in the following way while merging arrays B and C:

- If $head(B) \leq head(C)$, we append $head(B)$ to the merged array and remove it from array B as usual.
- If $head(B) > head(C)$, we append $head(C)$ to the merged array, remove it from array C, and increment $I(B,C)$ by the number of elements remaining in array B.

Terms $I(B)$ and $I(C)$ are computed recursively. Counting the number of inversions does not increase the asymptotic time complexity of the merge sort.

I-4 (**CLRS 4.2-6**) How quickly can you multiply a $kn \times n$ matrix by an $n \times kn$ matrix, using Strassen's algorithms as a subroutine? Answer the same question with the order of input matrices reversed.

Let the input be $A = [A_1^T \ \cdots \ A_k^T]^T$ and $B = [B_1 \ \cdots \ B_k]$, where A_i and B_i are $n \times n$ submatrices. The product AB is a $kn \times kn$ matrix

$$AB = \begin{bmatrix} A_1B_1 & \cdots & A_1B_k \\ \vdots & \ddots & \vdots \\ A_kB_1 & \cdots & A_kB_k \end{bmatrix},$$

where each product A_iB_j can be computed in $\Theta(n^{\log_2 7})$ time using Strassen's algorithm. There are k^2 such products, so the total time requirement is $\Theta(k^2 n^{\log_2 7})$.

The product BA is a $n \times n$ matrix $BA = \sum_{i=1}^{k} A_iB_i$. There are k products requiring $\Theta(n^{\log_2 7})$ time and $k-1$ summations requiring $\Theta(n^2)$ time. The total time is thus $\Theta(kn^{\log_2 7})$.

I-5 (**CLRS 4.2-7**) Show how to multiply the complex numbers $a+bi$ and $c+di$ using only three multiplications of real numbers. The algorithm should take a, b, c, and d as input and produce the real component $ac-bd$ and the imaginary component $ad+bc$ separately.

Compute the products ac, bd and $(a+b)(c+d)$. Now the real component is $ac-bd$ and imaginary component is $(a+b)(c+d)-ac-bd$.

Alternatively, compute e.g. $a(c-d)$, $d(a-b)$ and $b(c+d)$. Now the real component is obtained as the sum of the first two, and the imaginary component as the sum of the last two.

Both solutions use 3 multiplications and 5 additions/substractions.