Algorithms - CS5800 - Assignment 1

1. Recall that:

   - $f(n) = O(g(n))$ if there exist constants $N > 0$, $c > 0$ such that for any $n \geq N$ we have $0 \leq f(n) \leq c \cdot g(n)$.
   - $f(n) = o(g(n))$ if for any $c > 0$ there exists $N > 0$, such that for any $n \geq N$ we have $0 \leq f(n) \leq c \cdot g(n)$.
   - $f(n) = \Omega(g(n))$ if there exist constants $N > 0$, $c > 0$ such that for any $n \geq N$ we have $0 \leq c \cdot g(n) \leq f(n)$.
   - $f(n) = \Theta(g(n))$ if there exist constants $N > 0$, $c_1 > 0$, $c_2 > 0$ such that for any $n \geq N$ we have $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.

   Answer the following question:
   (a) Does $2^{n+1} = O(2^n)$?
   (b) Prove that for $n \geq 10$ we have $2^n \geq n^3$ and deduce that $2^n = \Omega(n^3)$.
   (c) Show that for any $a, b > 0$ we have $(n + a)^b = \Theta(n^b)$.

   For the following, let $f(n)$ and $g(n)$ be two positive functions.

   (d) Prove or disprove: $\max\{f(n), g(n)\} = \Theta(f(n) + g(n))$.
   (e) Prove or disprove:
   $f(n) = \Theta(g(n)) \iff f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
   (f) Prove or disprove:
   $f(n) = o(g(n)) \iff f(n) = O(g(n))$ and $f(n) \neq \Theta(g(n))$.

2. Let $T(n)$ be a function defined by the recurrence relation.

$$T(n) = \begin{cases} 1, & n = 1; \\ T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 1, & n > 1 \end{cases}$$

   Prove that $T(n) = O(n)$ (Do not use the Master theorm).

3. Use the Master theorem to find the asymptotic behavior of the function $T(n)$ in the following cases:
   (a) $T(n) = 4T(\lfloor \frac{n}{2} \rfloor) + 7n$, $T(1) = 1$.
   (b) $T(n) = 4T(\lfloor \frac{n}{2} \rfloor) + 5n^2$, $T(1) = 1$.
   (c) $T(n) = 4T(\lfloor \frac{n}{2} \rfloor) + 12n^3$, $T(1) = 1$.

4. If x equals one of the elements of the **sorted** array $A = (a_1, a_2, ..., a_n)$ (here $a_1 < a_2 < ... < a_n$) then we can find x using binary search in the following way:

   - Compare $a_{\frac{n}{2}}$ to x ($\frac{n}{2}$ may be $\lfloor \frac{n}{2} \rfloor$ for example).
   - If $a_{\frac{n}{2}} < x$ use binary search to look for x in $(a_{\frac{n}{2}+1}, ..., a_n)$.
   - If $a_{\frac{n}{2}} > x$ use binary search to look for x in $(a_1, ..., a_{\frac{n}{2}})$.

(a) Write pseudo code for an algorithm BinarySearch(A, n, x) that gets a sorted array A of length n and an element x as input, and returns the index of x in the array. (What happens if x is not in the array?)
Make sure that your algorithm performs $O(\log(n))$ comparisons in the worst case (prove it).
(b) Use binary search to improve the insertion sort algorithm that we described in class, so that it can sort an array using $O(n \log(n))$ comparisons in the worst case.

5. Describe a comparison algorithm that gets an array of numbers $A = (a_1, a_2, ..., a_n)$ and a real number x as input, and checks whether there are 2 elements in the array A whose sum equals x.
For example: if $A = (1, 4, 2, 7, 10, 11)$ and $x = 18$, the algorithm should return Yes (and possibly the indices 4 and 6).
If $A = (1, 4, 2, 7, 10, 11)$ and $x = 16$ the algorithm should return No.
You should find an algorithm with complexity $O(n \log_2(n))$ and prove it.
Write the pseudo code of your algorithm.

6. We saw that we can find the minimal element of an array of length n using $n-1$ comparisons. It immediately follows that we can find the first two smallest elements using $2n - 3$ comparisons. We want to do better.
Describe a comparison algorithm that gets an array $A = (a_1, a_2, ..., a_n)$ and returns the index of the minimal element and the index of the second smallest element using no more than $n + \log_2(n)$ comparisons (Note: this is not $O(n + \log(n))$ (which is just $O(n)$) but exactly $n + \log(n)$). You may assume that $n = 2^k$.
You do not have to write a pseudo code for the algorithm but you must explain how the algorithm works in a clear way that would enable its implementation.
Prove that the algorithm you described performs at most $n + \log_2(n)$ comparisons.