B1 Algorithms (25 points)

a. [6 points] State the solutions to the following recurrences using asymptotic notation.
$T(n) = 2T(n/2) + n$
$T(n) = 2T(n/2) + 1$
$T(n) = 4T(n/2) + n$
$T(n) = 2T(n/2) + n \lg(n)$

b. [4 points] What is the minimum number of comparisons to sort 7 integers?

c. [5 points] You have a roster of all students at the university and a list of students who have paid their tuition. Describe a fastest possible method to find out who have not paid yet, and give the worst-case complexity of your method. (Assume that the students are identified by their 9-digit SSN in both lists, which are in a random order).

d. [10 points] Compute the length of the longest common subsequence of two strings using Dynamic Programming technique.

i. Give the recurrence equation.
ii. Write your algorithm in a memoized style.
iii. Give the time complexity.

B2 Algorithms (25 points)

Select any 7 of the following 9 items (do only 7 -- if you do more, only the first 7 will be graded). For each item that you select, give a short explanation of the item. You should capture the main idea/ideas of the data structure, algorithm, or technique in your explanation. For algorithms and important operations of data structures, state the worst case running time and, if appropriate, average case, expected case, or amortized times.

1. Binomial Heap
2. Hash table with double hashing
3. Selection (worst case linear time)
4. Dijkstra's shortest path algorithm
5. Floyd-Warshall transitive closure algorithm
6. Convex hull algorithm (Graham scan or package wrap)
7. Strassen's matrix multiplication.
8. FFT
9. FFT based polynomial multiplication.

B3 Algorithms (25 points)

Part I: Consider this code for "introspective sort."

```
introspectiveSort(b, e)
// The items A[b]..A[e-1] will be permuted into sorted order.
// (indexing to a global array A is assumed throughout)
{
    introSort(b, e, ceiling(2*log(e-b)));
}


introSort(b, e, height)
// The items A[b]..A[e-1] will be permuted into sorted order.
// Requires integer height ≥ 0.
{   n = e - b;
    if ( n ≤ 16 )
        insertionSort(b, e);
    else if (height == 0)
        heapSort(b, e);
    else
    {
        q = partition(b, e); // partition as in quickSort.

        /* At this point, q is determined and elements are moved
         * around so that b < q < e and, for all i, j,
         * if b ≤ i < q ≤ j < e, then A[i] < A[j].
         */

        introSort(b, q, height-1);
        introSort(q, e, height-1);
    }
}
```

a. [4 points] Explain how insertion sort works and give it's worst case running time for an n item array.

b. [4 points] Explain how heap sort works and give it's worst case running time for an n item array.

c. [4 points] Explain why introspective sort correctly sorts and give it's worst case running time for an n item array. Assume partition running time is O(n), for n = e-b.

d. [4 points] Why is introspective sort preferable to either insertion sort or heap sort alone?

Part II: Disjoint set data structure (union-find).

e. [5 points] Explain the union-find data structure with path compression and union by rank. Discuss it's running time.

f. [2 points] What is the worst case running time if union by rank is used, but not path compression? Explain.

g. [2 points] Name an application of union-find.

B4 Algorithms (25 points)

a. [6 points] Define the problem classes P, NP and NP-complete.

b. The **partition problem** is, given a set of n nonnegative integers as input, to find a way to partition this set into two disjoint subsets so that the sums of the integers in each of the two subsets are equal. The **subset sum problem** is, given a set of n nonnegative integers and an integer k to find a subset such that the sum of the integers in the subset is equal to k.

   i. [2 points] Prove that the **partition problem** is in NP.

   ii. [8 points] If it is known that the **subset sum problem** is NP-complete, prove that the **partition problem** is also NP-complete, by using Polynomial Reduction technique. (Hint: Assume there is an algorithm that solves the **partition problem**, and try to use that algorithm as a subroutine to solve the **subset sum problem**).

   iii. [3 points] If an algorithm is designed by using dynamic programming technique such that it can solve the **subset sum problem** in time $O(kn^c)$, where c is a constant, will we be then able to claim that P = NP? Why or why not?

c. [6 points] Name and briefly describe four *different* NP-complete problems (the four problems should NOT be variations of the same problem, nor should any of them involve the **partition** and **subset sum** problems).