1. (20 pts) Each of the two sums below defines a function of $x$. Beneath each sum there appears a list of four assertions about the rate of growth, as $x \to \infty$, of the function that the sum defines. In each case, state which of the four choices, if any, are true (note: more than one choice may be true).

$$f_1(x) = \sum_{j \leq \sqrt{x}} \frac{1}{\sqrt{j}}$$

(1a) $O(\sqrt{x})$ \hspace{1cm} (1b) $o(x^{1/4})$ \hspace{1cm} (1c) $\sim 2x^{1/4}$ \hspace{1cm} (1d) $\sim 2x^{1/2}$

$$f_2(x) = \sum_{j \leq x} (\log j + j)$$

(2a) $O(x^2)$ \hspace{1cm} (2b) $O(x)$ \hspace{1cm} (2c) $O(x\log x)$ \hspace{1cm} (2d) $\sim x^2/2$

Solution of part 1 (10 pts):

Since $f_1(x)$ is not non-decreasing, we cannot apply Theorem 1.1.1 directly. However, if we ‘graph’ the rectangles representing the summation (as done in Fig. 1.1.1 and Fig. 1.1.2), we will get the following inequality:

$$\int_{\sqrt{x}}^{\sqrt{x}} t^{-1/2} \, dt \leq \sum_{j \leq \sqrt{x}} \frac{1}{\sqrt{j}} \leq \int_{0}^{\sqrt{x}} t^{-1/2} \, dt$$

$$\int_{0}^{\sqrt{x}} t^{-1/2} \, dt = 2(t^{1/2}) \bigg|_{1}^{\sqrt{x}} = 2(x^{1/2} - 1)^{1/2}$$

Therefore, (1a) is true.

$$\lim_{x \to \infty} \frac{2x^{1/4} - 2}{2x^{1/4}} = \lim_{x \to \infty} (1 - 1/x^{1/4}) = 1$$

Similarly, $\lim_{x \to \infty} \frac{2(x^{1/2} - 1)^{1/2}}{2x^{1/4}} = 1$, which means that (1c) is also true.

Solution of part 2 (10 pts):

By theorem 1.1.1, we have the following inequality:

$$\int_{0}^{x} (\log t + t) \, dt \leq \sum_{j \leq x} (\log j + j) \leq \int_{1}^{x+1} (\log t + t) \, dt$$

As shown in Fig. 1.1.7 and Fig. 1.1.8:
\[
\int_0^x (\log t) \, dt \leq \sum_{j \leq x} (\log j) \leq \int_1^{x+1} (\log t) \, dt \quad \Leftrightarrow \\
x \log x - x \leq \sum_{j \leq x} (\log j) \leq (x+1) \log(x+1) - x \\
(1)
\]

Also,
\[
\int_0^x (t) \, dt \leq \sum_{j \leq x} (j) \leq \int_1^{x+1} (t) \, dt \\
\int_1^{x+1} (t) \, dt = \frac{1}{2}t^2 \bigg|_1^{x+1} = \frac{1}{2}(x+1)^2 - \frac{1}{2} = \frac{1}{2}(x^2 + 2x + 1) - 1/2 = \frac{1}{2}x^2 + x + \frac{1}{2} - \frac{1}{2} = \frac{1}{2}x^2 + x \\
(2)
\]

Thus, \(1/2x^2 \leq \sum_{j \leq x} (j) \leq 1/2x^2 + x \)

Combining (1) and (2), we get:
\[
x \log x - x + 1/2x^2 \leq \sum_{j \leq x} (\log j + j) \leq (x+1) \log(x+1) + 1/2x^2 \\
\]

Therefore, **(2a) is true.**

\[
\lim_{x \to \infty} \frac{2\log x - x + 1/2x^2}{1/2x^2} = \lim_{x \to \infty} \frac{2\log x}{x} - 2/x + 1 = \lim_{x \to \infty} \frac{2\log x}{x} + \lim_{x \to \infty} (1-2/x) = \\
\lim_{x \to \infty} 2/(x\ln 2) + \lim_{x \to \infty} (1-2/x) = 0 + 1 = 1 \\
\]

Similarly, \(\lim_{x \to \infty} ((x+1)\log(x+1) + 1/2x^2)/1/2x^2 = 1\), which means that **(2d) is true.**

2. (25 pts)
(a) What is a bipartite graph? Give a definition.
(b) Devise an algorithm that decides if a given graph, of \(n\) vertices and \(m\) edges is or is not bipartite.
(c) What is the complexity of your algorithm.

**Solution of part a (5 pts):**

**Bipartite graph** is an undirected graph where vertices can be divided into two sets such that no edge connects vertices in the same set.

**Solution of part b (10 pts):**

**Pseudo code:**

(If the graph is bipartite, we will color the vertices in blue and red so that all edges have endpoints of different colors.)

Start at any vertex. Color it blue.
Pick an uncolored vertex.
If it is adjacent to a red vertex and a blue vertex.
Stop and output “The graph is not bipartite.”
If it is not adjacent to a blue vertex, color it blue.
If it is adjacent to a blue vertex, color it red.
If all vertices are colored, output “The graph is bipartite.”
Code:
The idea behind the algorithm is to look at all vertices (nodes) and try to separate them into 2 sets, such that there are no edges \((a,b)\), where \(a\) and \(b\) are in the same set.

\[
\text{set } s_1 = \{\}
\]
\[
\text{set } s_2 = \{\}
\]
global input: edges
global input: nodes

```
main()
    nodesToDo = nodes;
    while(nodesToDo.hasMoreItems()){
        pick n in nodesToDo;
        handle(n, true);
        breadthFirstSearch(n, !true);
        nodesToDo = nodesToDo - s1;
        nodesToDo = nodesToDo - s2;
    }
    HALT(true);
}
```

```
breadthFirstSearch(node n, boolean expectingS1){
    foreach neighbor n' in neighbors(n)
        handle(n', expectingS1);

    foreach neighbor n' in neighbors(n)
        if(toVisit(n)) breadthFirstSearch(n', !expectingS1);
}
```

// parameters: node n and a boolean value that defines whether we want node n
// to be in set s1 or s2
handle (node n, boolean expectingS1){
    // if we want n to be in s1, and it is already in s2, then the graph is not bipartite
    if(expectingS1 and n in s2) HALT(false);

    // if we want n to be in s2, and it is already in s1, then the graph is not bipartite
    if(!expectingS1 and n in s1) HALT(false);

    // if we visited this node already, we don’t want to do it again
    if(expectingS1 and n in s1) toVisit(n) = false;

    // otherwise, we do
    else if(expectingS1) toVisit(n) = true;

    // if we visited this node already, we don’t want to do it again
```
if(!expectingS1 and n in s2) toVisit(n) = false;

// otherwise, we do
else if(!expectingS1) toVisit(n) = true;

// add the node to the desired set
if(expectingS1) s1.add(n);
if(!expectingS1) s2.add(n);
}

Solution pf part c (10 pts):

When we visit a given node, we call handle() at most v-1 times, where v is the number of vertices in the graph, since in the worst case, every node can be connected to every other node, and there are v-1 of them. However, we will be “handling” the node twice – once as a neighbor, and once as the actual node. This means that for each node in the graph, we will be calling handle() 2(v-1) times. Therefore, the total number of calls to handle() is 2v(v-1).

Since in the worst case, every node is connected to every other node in the graph, we have that v^2 = e, where e is the number of edges in the graph. Thus, the complexity of the proposed algorithm is O(2v(v-1)) = O(v^2) = O(e).

3. (25 pts)
(a) What is the chromatic number of a cycle?
(b) Devise an algorithm that decides if a given graph contains an odd cycle.
(c) What is the complexity of your algorithm.

Solution of part a (5 pts):

**Chromatic number** of a graph G is the fewest number of colors we can use to color the vertices of G so that adjacent vertices have different colors.

**Proper coloring** of a graph G is an assignment of colors to vertices of G, such that adjacent vertices of G are assigned different colors.

**Cycle** is a graph whose vertices form a path <v_0, v_1, ..., v_{k-1}, v_k>, where v_0 = v_k.

An even cycle has chromatic number 2 since we can color all even nodes in blue and odd nodes in red. Such a coloring is proper.

An odd cycle has chromatic number 3. To see this, start from coloring v_0 blue, then v_1 must be red. If an even node is blue, then an odd node must be blue. However, for an odd cycle, v_0 and the last node have the same blue. So this is not a proper coloring. This shows that an odd cycle has chromatic number 3.
Solution of part b (10 pts):

A graph has a cycle iff there are two different paths from a vertex $u$ to another vertex $v$. This means having a cycle is equivalent to some vertex being visited twice by depth-first search. To see if a cycle is of odd length, we mark the vertices on the cycle alternately as odd and even. If the starting vertex is marked both odd and even, then the cycle has odd length.

global hasoddcycle = false;

// $L[]$ is an adjacency list representing a graph
// $n$ is the number of vertices
boolean hasOddCycle(int n, list L[]) {
    for (i = 1; i <= n; ++i) {
        visited[i] = unseen;
        parent[i] = 0;
    }
    visited[0] = 0;

    for (i = 1; i <= n; ++i) {
        if (visited[i] == unseen) {
            visit(i, L);
        }
    }
    return (!hasoddcycle);
}

void visit(int i, list L[]) {
    // alternating the markings of vertices
    visited[i] = 1 - visited[parent(i)];

    for (p = L[i]; p != NULL; p = next(p)) {
        // next vertex adjacent to $i$
        v = vertex(p);

        if (v == parent(i)) {
            continue;
        }

        if (visited[v] == visited[i]) { // $v$ has been seen already
            hasoddcycle = true;
        } else {
            parent(v) = i;
            visit(v, L);
        }
    }
}
Solution of part c (10 pts):

The running time is that of depth-first search, $O(v+e)$, where $v$ is the number of vertices in the graph, and $e$ is the number of edges.

4. (30 pts) MergeSort is a recursive sorting procedure that sorts an array of $n$ elements as follows:
   - if $n \leq 1$, then the array is already sorted. Stop now.
   - Otherwise, $n > 1$,
     - MergeSort the left half of the array,
     - MergeSort the right half of the array,
     - Merge the now-sorted left and right halves.

Let $T(n)$ denote the number of comparisons it takes in the worst case for MergeSort an array of $n$ element. Then $T(n)$ satisfies the following recurrence relation:

$$T(n) \leq T(n/2) + T(n/2) + n; \text{ if } n > 1; \text{ and } T(n) = 0, \text{ if } n \leq 1.$$  

(a) Prove that $T(n) = O(n \log n)$.
(b) What is the complexity of QuickSort? Compare it with MergeSort.

Solution of part a (15 pts):

Proof #1.

We want to show that $T(n) < n \log n$ where $\log$ is of base 2. This is true for $n \leq 1$.

Suppose this is true for $n' < n$.

We have $T(n) \leq 2T(n/2)) + n$

$$\leq 2n/2 \log (n/2) + n$$

$$\leq n (\log n - 1) + n$$

$$\leq n \log n$$

The proof is complete.

Proof #2

$$\begin{array}{ccccccc}
\text{cn} & \text{cn/2} & \text{cn/2} & \text{cn/4} & \text{cn/4} & \text{cn/4} & \text{cn/4} \\
\end{array}$$

Now, we just need to sum all the elements in the above recursion tree. Note, the sum of the elements in each row is $cn$, where $c$ is a constant. Since there are $\log n$ such rows, the sum over the entire tree is $cn \log n$, which implies that the complexity of MergeSort is $O(n \log n)$.

Solution of part b (15 pts):
Worst-case time complexity of QuickSort is $O(n^2)$ in case the array we are trying to sort is already sorted. As was shown in class, average-time complexity of QuickSort is $O(n \log n)$.

Clearly, in the worst case, MergeSort is doing better than QuickSort. However, in the average case, both do the same.