1. (10 marks)

Give the recurrences for the asymptotic worst-case time complexities of the following recursive algorithms:

a) (5 marks)

```plaintext
procedure FUNKY-REC(m, n)
    if (n < 10)
        sum = 0
        for i = 1 to m * n do
            sum = (sum + FUNKY-ITR(n)) - j
        return(sum)
    else if (n > 8)
        sum = FUNKY-REC(m, n - 9) + FUNKY-ITR(m)
        return(sum)
    else if (n > 15)
        sum = 0
        for i = 1 to n * n * n do
            sum = FUNKY-ITR(n) * l
        return(sum)
    else
        sum = 0
        for i = 1 to m * log(m) do
            l = FUNKY-REC(m, n - 12)
            sum = sum + (FUNKY-ITR(m) / i) - l
        return(sum)
```

Assume that procedure FUNKY-ITR(x) runs in $O(x^2 \log x)$ time.

$$T(m, n) = \begin{cases} 
O(m) & \text{if } n \leq 9 \\
T(m, n - 9) + O(m^2 \log m) & \text{otherwise}
\end{cases}$$
b) (5 marks)

```plaintext
procedure FUNKY-REC(m, n)
    if (n > 15)
        for i = 1 to log(m) do
            sum = FUNKY-REC(m, n / 9) + (FUNKY-ITR(m) - 12)
        return(sum)
    sum = 0
    for i = 1 to m do
        sum = sum + (FUNKY-ITR(n) / i) - j
    return(sum)
else if (n < 10)
    sum = 0
    for i = 1 to log(m) do
        sum = sum + (FUNKY-ITR(n) / i) - j
    return(sum)
    sum = FUNKY-REC(m, n - 9) - FUNKY-ITR(m)
    return(sum)
else
    sum = 0
    for i = 1 to m do
        l = FUNKY-REC(m, n - 6)
        sum = sum + (FUNKY-ITR(log(m)) / i) - l
    return(sum)
```

Assume that procedure FUNKY-ITR(x) runs in $O(x^2)$ time.

$$T(m, n) = \begin{cases} 
O(\log_2 m) & \text{if } n \leq 9 \\
T(m, n - 6) + O(m \log_2 m)^2 & \text{if } 10 \leq n \leq 15 \\
(\log_2 m)T(m, n/9) + O(m^2 \log_2 m) & \text{otherwise}
\end{cases}$$
2. (10 marks)
Derive the asymptotic worst-case time complexities of the following algorithms:

a) (5 marks)

```
sum = 42

csum = -57 + log(n)
for i = 1 to n do
    if (csum < sum)
        for j = 1 to log(n) do
            sum = sum - csum + j + j
        if (sum > csum)
            for k = 1 to log(n) do
                sum = sum - (k * k)
        else
            csum = sum - FUNKY(n * log(n) * 57)
    else
        for k = 1 to n * log(n) do
            csum = csum + sum / k
```

Assume that procedure FUNKY(x) runs in $O(x^2)$ time.

**Answer:** In the worst case, the innermost if-then-else statement requires max($O(\log n), O(n^2(\log n)^2)$) = $O(n^2(\log n)^2)$ time, and the outermost if-then-else statement requires max($O(n^2(\log n)^3), O(n \log n)$) = $O(n^2(\log n)^3)$ time. As this outermost if-then-else statement is embedded inside a for-loop that executes $n$ times, the algorithm as a whole requires $O(n \times n^2(\log n)^3) = O(n^3(\log n)^3)$ time.
b) (5 marks)

```plaintext
sum = 57 * n * n
csum = 10000001
if (sum * sum > csum)
    for i = 1 to n do
        if ((sum + i) > (sum * n * log(n))]
            sum = FUNKY(n / i) + FUNKY(n * n * n)
        else
            for j = 1 to log(n) do
                for k = 1 to log(n) do
                    sum = sum / i + k
                csum = csum + k / j
            else if (csum < sum)
                for k = 1 to n * 100000000 do
                    csum = csum / sum + k - FUNKY(log(n))
```

Assume that procedure FUNKY(x) runs in $O(x)$ time.

**Answer:** In the worst case, the innermost if-then-else statement requires max($O(n^3), O((\log_2 n)^2)) = O(n^3)$ time. Hence, the time required by the outermost if-the-else statement (and hence the algorithm as a whole) is max($O(n \times n^3), O(n \times \log_2 n)) = O(n^4)$ time.

3. (12 marks) Circle the letters associated with the appropriate answers in the following multiple choice questions. Note that some of these questions may have more than one answer whose letter needs to be circled.

(a) (6 marks) Which of the following is true of the function $T(n) = \frac{17}{4}n^{-3}$?

a) $T(n)$ is $O((n-3)^2)$  
   b) $T(n)$ is $\Theta(57 \log_2 2^n)$  
   c) $T(n)$ is $\Omega((\frac{n^2}{n})^2)$

**Answer:**

- **$T(n)$ is $O((n-3)^2)$**: This can be rewritten as $\frac{17}{4}n^{-3} = \frac{17}{4n^3} \leq c(n^2 - 6n + 9) = c(n - 3)^2$, which holds for $c = 1$ and $n_0 = 4$.
- **$T(n)$ is not $\Theta(57 \log_2 2^n)$**: This can be rewritten as $\frac{17}{4n^3} \leq c_1 57n$ (the big-Oh part) and $\frac{17}{4n^3} \geq c_2 57n$ (the big-Omega part) The former inequality holds for $c_1 = n_{0,1} = 1$; however, as the latter can be rewritten as $\frac{17}{228n^3} \geq c_2$ and $\frac{17}{228n^3}$ goes to zero as $n$ goes to infinity, the latter inequality fails for any constant $c_2$ for sufficiently large values of $n$.
- **$T(n)$ is $\Omega((\frac{n^2}{n})^2)$**: This can be rewritten as $\frac{17}{4n^3} \geq \frac{c}{n^n} = c(\frac{n^2}{n})^2 \Rightarrow \frac{17}{4n} \geq c$, which holds for $c = n_0 = 1$.

(b) (6 marks) Which of the following is true of the function $T(n) = 37n^3$?
a) $T(n)$ is $O(n^{\log_2 7})$

b) $T(n)$ is $\Theta(3^{\log_9 n})$

c) $T(n)$ is $\Omega(\sqrt{n})$

Answer:

- $T(n)$ is not $O(n^{\log_2 7})$: This can be rewritten as $37n^3 \leq cn^{\log_2 7}$. As $\log_2 7 < 3$, this can be rewritten as $37n^{c'} \leq c$ for $c' = 3 - \log_2 7 > 0$. As $37n^{c'}$ goes to infinity as $n$ goes to infinity, this inequality fails for any constant $c$ for suitably large values of $n$.

- $T(n)$ is not $\Theta(3^{\log_9 n})$: This can be rewritten as $37n^3 \leq c_1 n^{\frac{1}{2}} = c_1 n^{\log_3 n}$ (the big-Oh part) and $37n^3 \geq c_2 n^{\frac{3}{2}}$ (the big-Omega part). The latter inequality holds for $c_2 = n_{0.2} = 1$; however, as the former inequality can be rewritten as $37n^{2.5} \leq c_1$ and the quantity $37n^{2.5}$ goes to infinity as $n$ goes to infinity, the former inequality fails for any constant $c_1$ for suitably large values of $n$.

- $T(n)$ is not $\Omega(\sqrt{n})$: This can be rewritten as $37n^3 \geq c n^{\frac{3}{2}} = c (n^7)^{\frac{1}{2}} = \sqrt{n^7}$. As this can in turn be rewritten as $\frac{37}{n} \geq c$ and the quantity $\frac{37}{n}$ goes to zero as $n$ goes to infinity, the inequality fails for any constant $c$ for suitably large values of $n$.

4. (14 marks)

Consider the following algorithm:

```python
DFS-V(i, sol, U, B, VL, C)
    if (SIZE(sol) > B)
        print("pruned: solution too big for knapsack")
    else if (i == n + 1)
        if (VALUE(sol) < VL)
            print("pruned: solution too cheap")
        else if (VALUE(sol) <= (C * SIZE(sol)))
            print(sol)
    else
        DFS-V(i + 1, sol, U, B, VL, C)
        DFS-V(i + 1, UNION-COPY(sol, U[i]), U, B, VL, C)
```

The problem solved here is a variant of 0/1 KNAPSACK that prints all viable knapsack-loads of value $val$ and size $size$ such that $VL \leq val \leq C \times size$. In this algorithm, $U$ is the set of items, $n$ is the number of items in $U$, $B$ is the knapsack size bound, $sol$ is a subset of $U$, $VL$ is a lower bound on the summed value of the items in a solution, $C$ is a non-zero positive integer, $SIZE(sol)$ returns the sum of the sizes of the items in $sol$, $VALUE(sol)$ returns the sum of the values of the items in $sol$, and $UNION-COPY(sol, U[i])$ returns a copy of $sol$ to which the $i$th item in $U$ has been added. To answer this question, sketch the implicit tree of solution-nodes generated by the algorithm above (marking leaf-nodes which are printed by a circle and nodes that are
pruned with a square) with the call DFS-V(1, sol, U, B, VL, C) when \( U = \{X, Y, Z\} \) such that \( \text{size}(X) = 1, \text{size}(Y) = 4, \text{size}(Z) = 2, \text{value}(X) = 4, \text{value}(Y) = 2, \) and \( \text{value}(Z) = 3, \) \( \text{sol} \) is the empty set, and \( B, VL, \) and \( C \) have the following values:

i) (7 marks) \( B = 3, VL = 3, \) and \( C = 3: \)

\[
\begin{array}{ccc}
1 & 2 & 7 \\
& 3 & 6 & 8 \\
& & 4 & 5 & 9 & 10 \\
& & & 7 & 11 & 12 & 13
\end{array}
\]

ii) (7 marks) \( B = 4, VL = 2, \) and \( C = 1: \)

\[
\begin{array}{ccc}
1 & 2 & 9 \\
& 3 & 6 & 10 \\
& & 4 & 5 & 11 & 12 \\
& & & 13
\end{array}
\]

5. (14 marks) Consider the dynamic programming algorithm for the Matrix Chain Parentheses problem defined by the recurrence

\[
m[i, j] = \begin{cases} 
0 & \text{if } i = j \\
\min_{i \leq k < j} m[i, k] + m[k + 1, j] + p_{i-1}p_jp_k & \text{otherwise}
\end{cases}
\]

where the input is a matrix-chain dimension list \( p = \langle p_0, p_1, \ldots, p_n \rangle \) specifying a matrix-chain \( M_1M_2 \ldots M_n \) with dimensions \((p_0 \times p_1), (p_1 \times p_2), \ldots, (p_{n-1} \times p_n)\). Given matrix-chain dimension list \( p = \langle 3, 4, 2, 5, 3 \rangle \), compute and output an optimal parenthesization of the corresponding matrix-chain relative to the recurrence above (that is, fill in the given dynamic programming table (including one backpointer per table-cell), show the backpointer “path” that gives an optimal parenthesization, and give the parenthesization associated with that backpointer-“path”).
(( M1  M2 ) ( M3  M4 ))