Practice Midterm Examination

SCIS 313: Data Structures and Algorithm Analysis

Prof. Antonio Khalil Moretti Spelman College

Full solutions included. Work each problem before reading the solution.

Problem	Topic	Points
1	Big-O: definitions, proofs, comparisons	15
2	Analyzing iterative code	25
3	Recurrence relations	30
4	Challenge: recursive analysis	20
5	Merge sort trace	10
	Total	**100**

Reference: Three-Case Recurrence Theorem. For recurrences of the form $T(n) = aT(n/b) + O(n^d)$ with $a \geq 1$, $b > 1$, $d \geq 0$:

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \quad \textbf{(Case 1: equal work each level)} \\ O(n^d) & \text{if } a < b^d \quad \textbf{(Case 2: combine step dominates)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \textbf{(Case 3: recursion dominates)} \end{cases}$$

Problem 1

15 points

Big-O: Definitions, Proofs, and Comparisons.

Part (a)

5 points

Prove that $3n^2 + 8n + 2 = O(n^2)$.

Definition. $f(n) = O(g(n))$ if $\exists$ constants $c > 0$, $n_0 > 0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

For $n \geq 1$: $n \leq n^2$ and $1 \leq n^2$, so

$$3n^2 + 8n + 2 \leq 3n^2 + 8n^2 + 2n^2 = 13n^2.$$

Choose $c = 13$, $n_0 = 1$. Then $3n^2 + 8n + 2 \leq 13n^2$ for all $n \geq 1$. $\square$

Part (b)

4 points

True or False: $5n^3 + n = \Theta(n^2)$

False.

$\Theta(n^2)$ requires $O(n^2)$, meaning $5n^3+n \leq c \cdot n^2$ for some constant c. Divide by n^2: $5n+1/n \leq c$. The left side grows without bound, so no constant c works. Therefore $5n^3+n \neq O(n^2)$, and hence $\neq \Theta(n^2)$.

The correct bound is $\Theta(n^3)$. $\square$

Part (c)

6 points

Rank from slowest to fastest: $n^2,\ \log_2 n,\ n!,\ \sqrt{n},\ n\log_2 n,\ 2^n$

Then: Algorithm A is $O(n^2)$, Algorithm B is $\Theta(n \log n)$. For large n, which do you prefer?

$$\log_2 n \ \ll\ \sqrt{n} \ \ll\ n\log_2 n \ \ll\ n^2 \ \ll\ 2^n \ \ll\ n!$$

Key steps: logs grow slower than any positive power of n; $n\log n \ll n^2$ because $\log n \ll n$; any polynomial is dominated by any exponential; $n! \gg 2^n$ because $n! = n \cdot (n-1) \cdots 1$ while $2^n = 2 \cdot 2 \cdots 2$, and for large n most factors of $n!$ exceed 2.

Prefer Algorithm B. $\Theta(n \log n)$ is a tighter (and better) bound than $O(n^2)$. Since $n \log n \ll n^2$, for large n B is provably faster. Note: $O(n^2)$ only gives an upper bound; B's Θ bound tells us its *exact* growth rate.

Problem 2

25 points

Analyzing Iterative Code.

Part (a)

6 points

```
int total = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < 5; j++) {
        total++;
    }
}
```

Outer loop: n iterations. Inner loop: exactly 5 per outer iteration (fixed, independent of i).

$$T(n) = 5n$$

Big-O: $5n \leq 5n$ for all $n \geq 1$. Choose $c = 5$, $n_0 = 1$: $T(n) = O(n)$. Also $5n \geq 5n$, so $T(n) = \Omega(n)$. Therefore $T(n) = \Theta(n)$.

A constant inner bound never changes the Big-O class.

Part (b)

8 points

```
int sum = 0;
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
        sum++;
    }
}
```

When $i = 0$: inner runs n times. When $i = 1$: $n - 1$ times. ... When $i = n - 1$: 1 time.

$$T(n) = \sum_{i=0}^{n-1}(n - i) = n + (n-1) + \cdots + 1 = \frac{n(n+1)}{2}$$

Big-O: $\frac{n(n+1)}{2} = \frac{n^2+n}{2} \leq \frac{n^2+n^2}{2} = n^2$ for $n \geq 1$. So $T(n) = O(n^2)$.

Lower bound: $\frac{n(n+1)}{2} \geq \frac{n^2}{2}$, so $T(n) = \Omega(n^2)$. Therefore $T(n) = \Theta(n^2)$.

Part (c)

11 points

```
// Section A: nested loops
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        cout << i + j;

// Section B: halving loop
```

```
7  int k = n;
8  while (k > 1) { k = k / 2; }
```

Section A: outer runs n times, inner runs n times each iteration. Total: $n \times n = n^2$ executions. $\Theta(n^2)$.

Section B: k is halved each iteration: $n \to n/2 \to n/4 \to \cdots \to 1$. After k iterations, value is $n/2^k$; stops when $n/2^k \leq 1$, i.e. $k = \lceil \log_2 n \rceil$. Total: $\Theta(\log n)$.

Overall: $T(n) = n^2 + \log n$. For large n, $n^2 \gg \log n$, so

$$T(n) = \Theta(n^2).$$

Section A dominates. Adding $\log n$ to n^2 is negligible: $n^2 + \log n \leq 2n^2$ for $n \geq 1$, so the overall bound is still $\Theta(n^2)$.

Problem 3

30 points

Recurrence Relations.

Part (a)

10 points

Solve by unrolling: $T(1) = c, \quad T(n) = T(n/2) + n$.

$$\begin{aligned} T(n) &= T(n/2) + n \\ &= T(n/4) + (n/2) + n \\ &= T(n/8) + (n/4) + (n/2) + n \\ &= T(n/2^k) + n\left(\frac{1}{2^{k-1}} + \cdots + \frac{1}{2} + 1\right) = T(n/2^k) + n\sum_{i=0}^{k-1}\left(\frac{1}{2}\right)^i \end{aligned}$$

Stop when $n/2^k = 1$, i.e. $k = \log_2 n$:

$$T(n) = c + n\sum_{i=0}^{\log_2 n - 1}\left(\frac{1}{2}\right)^i \leq c + n\sum_{i=0}^{\infty}\left(\frac{1}{2}\right)^i = c + n \cdot \frac{1}{1 - 1/2} = c + 2n$$

The geometric series with ratio $1/2$ converges to 2, so the whole sum is bounded by $2n$. Since the $i = 0$ term alone gives n, we also have $T(n) \geq n$.

$$T(n) = \Theta(n)$$

Intuition: the work $n, n/2, n/4, \ldots$ shrinks so fast that the root level dominates.

Part (b)

12 points

Apply the Three-Case Recurrence Theorem. Recall: $T(n) = aT(n/b) + O(n^d)$; compare a to b^d.

(i) $T(n) = 4T(n/2) + O(n)$

$a = 4$, $b = 2$, $d = 1$. $b^d = 2$. $a = 4 > 2 = b^d$ **Case 3.**

$$T(n) = O(n^{\log_2 4}) = O(n^2)$$

(ii) $T(n) = 3T(n/3) + O(n)$

$a = 3$, $b = 3$, $d = 1$. $b^d = 3$. $a = 3 = 3 = b^d$ **Case 1.**

$$T(n) = O(n^d \log n) = O(n \log n)$$

(iii) $T(n) = 2T(n/2) + O(n^2)$

$a = 2$, $b = 2$, $d = 2$. $b^d = 4$. $a = 2 < 4 = b^d$ **Case 2.**

$$T(n) = O(n^d) = O(n^2)$$

The quadratic combine step dominates; recursion barely contributes.

Part (c)

8 points

Student claims Three-Case Recurrence Theorem applies to $T(n) = 2T(n-1) + O(1)$.

(i) The student is wrong.

The Three-Case Recurrence Theorem requires recurrences of the form $T(n/b)$ — the subproblem size must be a *fraction* of n (divided by a constant $b > 1$). Here the subproblem size is $n - 1$ (subtraction, not division), so the Three-Case Recurrence Theorem does not apply.

(ii) Solve by unrolling:

$$\begin{aligned} T(n) &= 2T(n-1) + c \\ &= 2[2T(n-2) + c] + c = 4T(n-2) + 2c + c \\ &= 8T(n-3) + 4c + 2c + c \\ &= 2^k T(n-k) + c(2^{k-1} + \cdots + 2 + 1) = 2^k T(n-k) + c(2^k - 1) \end{aligned}$$

Stop when $n - k = 1$, i.e. $k = n - 1$:

$$T(n) = 2^{n-1}T(1) + c(2^{n-1} - 1) = 2^{n-1} \cdot c_0 + c(2^{n-1} - 1) = \Theta(2^n)$$

Exponential time — extremely slow.

Problem 4

20 points

Challenge: Analyzing a Recursive Function.

```
int f(int n) {
    if (n <= 1) return 1;
    int a = f(n / 4);
    int b = f(n / 4);
    int result = 0;
    for (int i = 0; i < n; i++) result += i;
    return a + b + result;
}
```

(a) Write the recurrence. Identify a, b, d.

Two recursive calls on $n/4$; $O(n)$ work outside.

$$T(1) = c \qquad T(n) = 2T(n/4) + O(n)$$

$a = 2, \quad b = 4, \quad d = 1.$

(b) Apply the Three-Case Recurrence Theorem.

$b^d = 4^1 = 4$. $a = 2 < 4 = b^d$ **Case 2.**

$$T(n) = O(n^d) = O(n)$$

The linear scan at the root dominates; the recursion contributes only a constant factor.

(c) Change `f(n/4)` to `f(n/2)`. New recurrence and analysis.

$$T(n) = 2T(n/2) + O(n)$$

$a = 2$, $b = 2$, $d = 1$. $b^d = 2$. $a = 2 = b^d$ **Case 1.**

$$T(n) = O(n \log n)$$

This is **worse** than the original $O(n)$. Making the recursion "shallower" (halving instead of quartering) actually increases the total work because each call now has a larger subproblem, and with $a = b^d$ the work is equal at every level rather than shrinking.

(d) Keep `f(n/4)` but reduce to 1 recursive call. New recurrence and analysis.

$$T(n) = T(n/4) + O(n)$$

$a = 1$, $b = 4$, $d = 1$. $b^d = 4$. $a = 1 < 4 = b^d$ **Case 2.**

$$T(n) = O(n^d) = O(n)$$

Same asymptotic bound as the original. Cutting from 2 calls to 1 does not change the

complexity here because the work is already dominated by the linear scan at the root.

Which has greater impact? Modification (c) — changing the subproblem size — has *worse* impact (worsens the bound from $O(n)$ to $O(n \log n)$). Modification (d) has no asymptotic impact. The key lesson: when the combine step dominates (Case 2), neither the number of calls nor the subproblem size matters much as long as we stay in Case 2. What matters is the exponent d of the outside work.

Problem 5

10 points

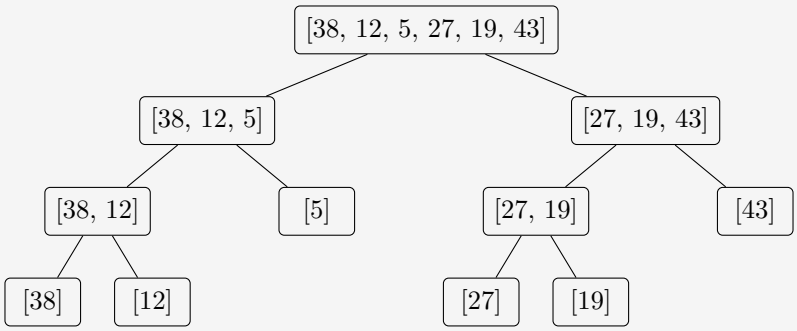
Merge Sort Trace.

Consider the array $A = [38,\ 12,\ 5,\ 27,\ 19,\ 43]$.

Part (a)

4 points

Draw the full **divide phase** of merge sort on A. Show each recursive split until every subarray has size 1.



Part (b)

4 points

Show the **merge phase** step by step, merging sorted subarrays back up until the full array is sorted. Write the result of each merge.

Working bottom-up:

Level 3: $[38], [12] \to [12, 38]$ $[27], [19] \to [19, 27]$

Level 2: $[12, 38] + [5] \to [5, 12, 38]$ $[19, 27] + [43] \to [19, 27, 43]$

Level 1: $[5, 12, 38] + [19, 27, 43] \to [5, 12, 19, 27, 38, 43]$

Final sorted array: $[5,\ 12,\ 19,\ 27,\ 38,\ 43]$.

Part (c)

2 points

The merge of $[5, 12, 38]$ and $[19, 27, 43]$ produces $[5, 12, 19, 27, 38, 43]$. How many element **comparisons** does this merge require? Justify your answer.

5 comparisons. Trace the two-pointer merge:

Step	Compare	Winner	Output so far	Advance
1	5 vs 19	5	$[5]$	i
2	12 vs 19	12	$[5, 12]$	i
3	38 vs 19	19	$[5, 12, 19]$	j
4	38 vs 27	27	$[5, 12, 19, 27]$	j
5	38 vs 43	38	$[5, 12, 19, 27, 38]$	i
Left exhausted; copy remaining right: append 43.				

In general, merging arrays of sizes p and q takes at most $p+q-1$ comparisons; here $3+3-1=5$.

Study tips for the real exam

- For Big-O proofs: write the definition first, then bound each extra term by a power of n.
- For Θ proofs: both directions required. Missing Ω is the most common error.
- For loop analysis: when the inner bound depends on the outer variable, write a summation.
- For unrolling: four steps $\rightarrow$ pattern $\rightarrow$ plug in stopping condition.
- For Three-Case Recurrence Theorem: $a \rightarrow b \rightarrow d \rightarrow b^d \rightarrow$ compare $\rightarrow$ case $\rightarrow$ result.
- When Three-Case Recurrence Theorem fails: check whether the recurrence subtracts rather than divides.