Assignment 4: Solutions

SCIS 313: Data Structures and Algorithm Analysis

Prof. Antonio Khalil Moretti Spelman College

Problem 1 — Setting Up Recurrences

(a) Split into two halves, recurse on both, $O(n)$ combine.

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

(b) Split into two halves, recurse on *one*, $O(1)$ outside.

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

(c) Three equal thirds, recurse on all three, $O(n^2)$ combine.

$$T(1) = c \qquad T(n) = 3T(n/3) + O(n^2)$$

(d) Reading the code: two recursive calls each on $n/4$; outside work is the addition, which is $O(1)$ (the `+ n` is a single addition, not a loop).

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

Common mistake: writing $O(n)$ for the `+ n` term. The `+ n` is arithmetic on a single integer, not a loop over n elements. The work outside the recursive calls is $O(1)$.

Corrected recurrence: $T(n) = 2T(n/4) + O(1)$.

Problem 2 — Solving by Unrolling

(a) $T(n) = T(n-1) + n, \quad T(1) = 1.$

Unroll:

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &= T(n-3) + (n-2) + (n-1) + n \\ &\vdots \\ &= T(1) + 2 + 3 + \cdots + n \end{aligned}$$

Sum the arithmetic series:

$$T(n) = 1 + \sum_{k=2}^{n} k = \sum_{k=1}^{n} k = \frac{n(n+1)}{2} = \Theta(n^2)$$

(b) $T(n) = T(n/4) + 1, \quad T(1) = c.$

Let $n = 4^m$ so $m = \log_4 n$. Unroll:

$$\begin{aligned} T(n) &= T(n/4) + 1 \\ &= T(n/16) + 1 + 1 \\ &= T(n/4^k) + k \end{aligned}$$

Stop when $n/4^k = 1$, i.e. $k = \log_4 n$:

$$T(n) = T(1) + \log_4 n = c + \log_4 n = \Theta(\log n)$$

(The base of the logarithm is a constant factor, absorbed by Θ.)

(c) $T(n) = 3T(n/3) + c, \quad T(1) = c.$

At level k: there are 3^k subproblems each of size $n/3^k$, each doing c work outside recursive calls. Work at level k: $3^k \cdot c$.

$$T(n) = c + 3c + 9c + \cdots + 3^k c + \cdots$$

The tree has $\log_3 n$ levels (stop when $n/3^k = 1$). At the leaves ($k = \log_3 n$): $3^{\log_3 n} = n$ leaf nodes each costing c.

$$T(n) = c \sum_{k=0}^{\log_3 n} 3^k = c \cdot \frac{3^{\log_3 n + 1} - 1}{3 - 1} = c \cdot \frac{3n - 1}{2} = \Theta(n)$$

The sum is a geometric series with ratio $r = 3 > 1$, so the last term dominates.

Problem 3 — Applying the Master Method

Recall: given $T(n) = aT(n/b) + O(n^d)$, compare a to b^d.

(a) $T(n) = 5T(n/2) + O(n^2)$.

$a = 5$, $b = 2$, $d = 2$. $b^d = 4$. $a = 5 > 4 = b^d$ **Case 3.**

$$T(n) = O(n^{\log_2 5}) \approx O(n^{2.32})$$

(b) $T(n) = 8T(n/2) + O(n^3)$.

$a = 8$, $b = 2$, $d = 3$. $b^d = 8$. $a = b^d$ **Case 1.**

$$T(n) = O(n^3 \log n)$$

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

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

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

Intuition: the combine step at the root dominates. Recursion adds nothing.

(d) $T(n) = 4T(n/3) + O(n)$.

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

$$T(n) = O(n^{\log_3 4}) \approx O(n^{1.26})$$

(e) $T(n) = 2T(n/2) + O(n \log n)$.

Master Method does not apply in the form given in class.

The combine term $n \log n$ is not of the form n^d for any constant d, because $\log n$ is not a polynomial factor. Formally, for any $d > 1$, $n \log n = o(n^d)$, and for $d = 1$, $n \log n \neq \Theta(n^1)$.

The answer (via Akra–Bazzi or careful tree analysis) is $T(n) = \Theta(n \log^2 n)$.

(f) $T(n) = T(n-1) + O(n^2)$.

Master Method does not apply. The recurrence subtracts a constant (1) from n rather than dividing by a factor $b > 1$. The Master Method requires $T(n/b)$ with b constant and $b > 1$.

Solve by unrolling:

$$T(n) = T(n-1) + cn^2 = T(1) + c\sum_{k=2}^{n} k^2 = c \cdot \frac{n(n+1)(2n+1)}{6} = \Theta(n^3)$$

Problem 4 — From Code to Recurrence to Solution

(a) **Parameters.**

The function makes **3** recursive calls, each on a subarray of size $n/3$, and then does a linear scan of length n.

$$a = 3, \qquad b = 3, \qquad d = 1$$

(b) **Recurrence.**

$$T(1) = c \qquad T(n) = 3T(n/3) + O(n)$$

(c) **Master Method.**

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

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

(d) **Is it useful?**

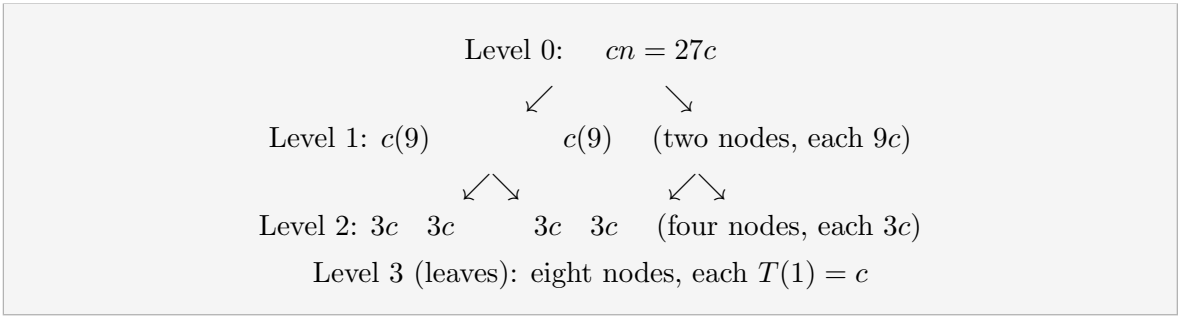
No. The function computes the sum of the array elements plus three recursive sub-sums — but since those sub-sums overlap and are summed together without correction, the result overcounts every element. Ignoring correctness, computing the array sum is trivially done in $O(n)$ with a single loop:

```
int sum = 0;
for (int i = lo; i <= hi; i++) sum += arr[i];
```

This is $O(n)$, which is strictly better than the $O(n \log n)$ of `mystery`.

Problem 5 — Recursion Tree for $T(n) = 2T(n/3) + cn$, $n = 27$

(a) **Tree.** (Each node shows the work done at that node, not the subproblem size.)



(b) **Table.**

Level k	Subproblem size	Nodes at level k	Work at level k
0	n	1	cn
1	$n/3$	2	$2c(n/3) = \frac{2cn}{3}$
2	$n/9$	4	$4c(n/9) = \frac{4cn}{9}$
k	$n/3^k$	2^k	$2^k \cdot c(n/3^k) = cn \cdot (2/3)^k$

(c) **Number of levels.**

Recursion stops when $n/3^k = 1$, i.e. $k = \log_3 n$ levels. For $n = 27$: $\log_3 27 = 3$ levels of recursion, so 4 levels total (0 through 3).

(d) **Total work.**

Sum across all levels (geometric series with ratio $r = 2/3 < 1$):

$$T(n) = \sum_{k=0}^{\log_3 n} cn \cdot \left(\frac{2}{3}\right)^k = cn \sum_{k=0}^{\infty} \left(\frac{2}{3}\right)^k = cn \cdot \frac{1}{1 - 2/3} = 3cn = O(n)$$

Since $r < 1$, the series is dominated by the root.

Verification via Master Method: $a = 2$, $b = 3$, $d = 1$. $b^d = 3$. $a = 2 < 3 = b^d \Rightarrow$ Case 2: $T(n) = O(n^1) = O(n)$. ✓

Problem 6 — Binary Search Correctness and Complexity

(a) **Integer overflow.**

If `lo` and `hi` are both large (close to `INT_MAX`), then `lo + hi` overflows a 32-bit signed integer, yielding a negative or otherwise incorrect value. The expression `lo + (hi - lo) / 2` avoids this: since `hi >= lo`, the subtraction `hi - lo` is non-negative and at most `INT_MAX`, so no overflow occurs.

(b) **Recurrence and solution.**

Each call makes one recursive call on half the array and does $O(1)$ comparisons:

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

Unroll:

$$T(n) = T(n/2) + c = T(n/4) + 2c = \cdots = T(n/2^k) + kc$$

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

$$T(n) = c + c \log_2 n = \Theta(\log n)$$

(c) **Trace on `[3,7,11,15,22,31,40,55,68,72]`, `target=31`.**

Call	`lo`	`hi`	`mid`	`arr[mid]` and decision
1	0	9	4	`arr[4]=22` $<$ 31 $\Rightarrow$ search right
2	5	9	7	`arr[7]=55` $>$ 31 $\Rightarrow$ search left
3	5	6	5	`arr[5]=31` $=$ 31 $\Rightarrow$ **found at index 5**

(d) **Maximum comparisons on 10 elements.**

$\lceil \log_2 10 \rceil = 4$.

After each comparison we either find the target or halve the search space. Starting from 10 elements: $10 \to 5 \to 2 \to 1$ (or $10 \to 5 \to 3 \to 1$ depending on rounding). In the worst case (element not present or found last), we need 4 comparisons before the search space collapses to `lo > hi`.

Problem 7 — Karatsuba Multiplication

(a) **Compute** 4321×8765**.** $a = 43$, $b = 21$, $c = 87$, $d = 65$. $n = 4$.

$$P_1 = a \times c = 43 \times 87 = 3741$$
$$P_2 = b \times d = 21 \times 65 = 1365$$
$$P_3 = (a+b)(c+d) = 64 \times 152 = 9728$$
$$\text{Cross term} = P_3 - P_1 - P_2 = 9728 - 3741 - 1365 = 4622$$

Combine (with $n = 4$, so pad by 10^4 and 10^2):

$4321 \times 8765 = 3741 \cdot 10^4 + 4622 \cdot 10^2 + 1365 = 37410000 + 462200 + 1365 =$ **37,873,565**

(b) **Naive recursive algorithm.**

$a = 4$, $b = 2$, $d = 1$ (four calls on $n/2$-digit numbers; $O(n)$ for additions and padding).

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

$b^d = 2$. $a = 4 > 2$ Case 3:

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

No improvement over the grade-school algorithm.

(c) **Karatsuba recurrence.**

$a = 3$, $b = 2$, $d = 1$.

$$T(n) = 3T(n/2) + O(n)$$

$b^d = 2$. $a = 3 > 2$ Case 3:

$$T(n) = O(n^{\log_2 3}) \approx O(n^{1.585})$$

The exponent is $\log_2 3 < 2$ because $3 < 2^2 = 4$. Reducing the number of recursive calls from 4 to 3 reduces a, which directly reduces $\log_b a$, the exponent in Case 3.

(d) **Hypothetical 2-call variant.**

If $a = 2$, $b = 2$, $d = 1$: $b^d = 2 = a$ Case 1:

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

This would be faster than Karatsuba. Whether it is achievable is not obvious — it would require computing three independent linear combinations of ac, $ad + bc$, and bd using only two multiplications of $n/2$-digit numbers, which would be a remarkable algebraic trick. No such algorithm is known for general integer multiplication. (For comparison,

the best known algorithm runs in $O(n \log n)$ via FFT-based methods, but those exploit structure beyond simple D&C splitting.)

Problem 8 — Algorithm Design: Two-Sum on a Sorted Array

(a) **Brute force.**

Check every pair (i, j) with $i < j$:

```
for i = 0 to n-1:
    for j = i+1 to n-1:
        if A[i] + A[j] == t: return true
return false
```

There are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs. Time complexity: $O(n^2)$.

(b) **Divide and conquer.**

Divide: split the array at the midpoint into $L = A[0 \ldots m]$ and $R = A[m{+}1 \ldots n{-}1]$.

Conquer: recursively solve two-sum on L alone (target t), and two-sum on R alone (target t).

Combine: additionally check for pairs that *straddle* the midpoint: for each $A[i] \in L$, binary-search R for $t - A[i]$. Since R is sorted, each binary search costs $O(\log n)$; there are $n/2$ elements in L, so the combine step costs $O(n \log n)/2 = O(n \log n)$.

Pseudocode for the cross-pair check:

```
for each A[i] in L:
    if binarySearch(R, t - A[i]) found: return true
```

(c) **Recurrence and solution.**

The cross-pair check dominates the combine step:

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

As noted in Problem 3(e), the Master Method does not directly apply here. The answer (by Akra–Bazzi or careful tree analysis) is $T(n) = O(n \log^2 n)$.

This is worse than $O(n^2)$ for small n but better asymptotically — though still not optimal.

(d) $O(n)$ **two-pointer algorithm.**

Initialize two pointers: $\ell = 0$ (left end), $r = n - 1$ (right end).

```
while l < r:
    s = A[l] + A[r]
    if s == t: return true
    else if s < t: l++    // need a larger sum
    else: r--             // need a smaller sum
return false
```

Why it works: at each step, if $A[\ell] + A[r] < t$ then no element to the left of $A[r]$ paired with $A[\ell]$ can reach t (since the array is sorted and $A[r]$ is already the largest available partner), so we can safely advance ℓ. Symmetrically for r. Every iteration advances at least one pointer, so the loop runs at most n times.

Time: $O(n)$. No extra space beyond the two pointer variables.