CS-3510-02 Algorithms, Spring 2025 Homework 4
Name: Hayden Hawley Collaborator(s):

Problem 1 (18 pts)

Suppose you are choosing between the following three algorithms:

(a) (5pts) Algorithm A solves the problem by dividing it into seven subproblems of half the size, recursively solves each subproblem, and then combines the solution in linear time.

(b) (5pts) Algorithm B solves the problem by dividing it into twenty-five subproblems of one fifth the size, recursively solves each subproblem, and then combines the solutions in quadratic time.

(c) (8pts) Algorithm C solves problems of size n by recursively solving four subproblems of size n-4, and then combines the solution in constant time.

In all cases you can assume it takes O(1) time to solve instances of size 1. What are the running times of each of these algorithms? Justify your answers. You can use the Master Theorem in your answers.

Solution

(a) **Algorithm A:**

Divide: Splits the problem into 7 subproblems, each of size $n/2$.
Conquer: Solves each subproblem recursively.
Combine: Merges the results in $O(n)$ time.

Recurrence Relation:

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n)$$

Master Theorem:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$$a = 7, \quad b = 2, \quad f(n) = O(n)$$

$$n^{\log_b a} = n^{\log_2 7} \approx n^{2.807}$$

Since $f(n) = O(n)$ is smaller than $n^{\log_2 7}$, this fits Case 1 of the Master Theorem

$$\boxed{T(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807})}$$

Therefore, overall time is worse than quadratic but a little better than cubic.

(b) **Algorithm B:**

Divide: Breaks the problem into 25 subproblems, each of size $n/5$.
Conquer: Solves each subproblem recursively.
Combine: Takes $O(n^2)$ to combine the results.

Recurrence Relation:

$$T(n) = 25T\left(\frac{n}{5}\right) + O(n^2)$$

Master Theorem:

$$a = 25, \quad b = 5, \quad f(n) = O(n^2)$$

$$n^{\log_b a} = n^{\log_5 25} = n^2$$

Since $f(n) = \Theta(n^2)$ matches $n^{\log_b a}$, this fits Case 2 of the Master Theorem:

$$\boxed{T(n) = \Theta(n^2 \log n)}$$

The recursion and the combining step are equally slow, so the overall time is slightly worse than quadratic.

(c) **Algorithm C:**

Divide: Splits into 4 subproblems, each of size $n - 4$.
Conquer: Recursively solves all 4.
Combine: Spends only constant time, or $O(1)$.

Recurrence Relation:

$$T(n) = 4T(n-4) + O(1)$$

Unroll the Recurrence:

$$\begin{aligned}
T(n) &= 4T(n-4) + O(1) \\
&= 4^2T(n-8) + 4O(1) + O(1) \\
&= 4^3T(n-12) + 4^2O(1) + 4O(1) + O(1) \\
&\dots \\
&= 4^kT(n-4k) + \sum_{i=0}^{k-1} 4^i \cdot O(1)
\end{aligned}$$

The sum is a geometric series, so

$$T(n) = \Theta(4^{n/4}) = \Theta\left(2^{n/2}\right)$$

Final Answer:

$$\boxed{T(n) = \Theta(2^{n/2})}$$

This grows exponentially. Since the problem size only shrinks by 4 each step but spawns 4 new calls, the total work grows very fast.

Problem 2 (10 pts)

Solve the following recurrences by unrolling the recurrence.

(a) (5 pts) $T(n) = 4T(n/3) + n^{3/2}$ for $n \geq 2; T(1) = 1$;

(b) (5 pts) $T(n) = T(3n/4) + n$ for $n \geq 2; T(1) = 1$;

Solution

(a) $T(n) = 4T(n/3) + n^{3/2}$

Unroll the recurrence:

$$\begin{aligned} T(n) &= 4T(n/3) + n^{3/2} \\ &= 4[4T(n/9) + (n/3)^{3/2}] + n^{3/2} \\ &= 4^2 T(n/9) + 4(n/3)^{3/2} + n^{3/2} \\ &= 4^2 T(n/9) + n^{3/2}(4 \cdot 3^{-3/2} + 1) \\ &\cdots \\ &= 4^k T(n/3^k) + n^{3/2} \sum_{i=0}^{k-1} 4^i \cdot 3^{-3i/2} \end{aligned}$$

The sum is geometric:

$$\sum_{i=0}^{k-1} \left(\frac{4}{3^{3/2}}\right)^i$$

Since $\frac{4}{3^{3/2}} < 1$, the sum converges to a constant.

$$T(n) = \Theta(1) + n^{3/2} \cdot \Theta(1) = \boxed{\Theta(n^{3/2})}$$

(b) $T(n) = T(3n/4) + n$

Unroll the recurrence:

$$\begin{aligned} T(n) &= T(3n/4) + n \\ &= T((3/4)^2 n) + (3n/4) + n \\ &= T((3/4)^3 n) + (3^2 n/4^2) + (3n/4) + n \\ &\cdots \\ &= T((3/4)^k n) + n \sum_{i=0}^{k-1} \left(\frac{3}{4}\right)^i \end{aligned}$$

The sum is geometric:

$$\sum_{i=0}^{k-1} \left(\frac{3}{4}\right)^i = \frac{1 - (3/4)^k}{1 - 3/4} = 4\left(1 - \left(\frac{3}{4}\right)^k\right)$$

So:

$$T(n) = T(1) + n \cdot \Theta(1) = \boxed{\Theta(n)}$$

Problem 3 (10 pts)

You are given two sorted lists of integers of length m and n (no ties exist and sorted in increasing order). Give an $O(\log m + \log n)$ time algorithm for computing the k-th smallest integer in the union of the lists.

Show the pseudo-code for your proposed algorithm and include any necessary justifications. **You need to analyze** the time complexity of your algorithm to show that it is $O(\log m + \log n)$

Solution

We use a binary search on the smaller array to find the k-th smallest element in the union of the two sorted arrays. Let the arrays be $A[0..m-1]$ and $B[0..n-1]$ and assume $m \leq n$.

- Take i elements from A
- Take $k - i$ elements from B
- We want the max of the left halves $\leq$ the min of the right halves

Pseudo code:

```
function find_Kth(A, B, k):
    if len(A) > len(B):
        return find_Kth(B, A, k)
    m = len(A)
    n = len(B)

    low = max(0, k - n)
    high = min(k, m)

    while low <= high:
        i = (low + high) // 2
        j = k - i

        A_left = A[i - 1] if i > 0 else -infinity
        A_right = A[i] if i < m else +infinity
        B_left = B[j - 1] if j > 0 else -infinity
        B_right = B[j] if j < n else +infinity

        if A_left <= B_right and B_left <= A_right:
            return max(A_left, B_left)
        elif A_left > B_right:
            high = i - 1
        else:
```

```
        low = i + 1
```

- Bbinary search on array A, checking if we can make a valid partition.
- If $A[i-1] > B[k-i]$, then we picked too many from A, so reduce i.
- If $B[k-i-1] > A[i]$, then we picked too few from A, so increase i.
- Stop when the left parts from both arrays are smaller than the right parts.
- If one partition is empty (e.g., $i = 0$ or $i = m$), we use $\pm\infty$ to avoid out-of-bounds errors.
- k must be between 1 and $m + n$

Time analysis: - Binary search on the smaller array A takes $O(\log m)$
- Each iteration computes constant-time comparisons.
- No search on B since $j = k - i$ is derived.
- Total runtime: $\boxed{O(\log m + \log n)}$

Problem 4 (10 pts)

Strassen's algorithm is a classic example of using the divide-and-conquer paradigm to reduce the asymptotic time complexity of matrix multiplication from $O(n^3)$ to approximately $O(n^{\log_2 7}) \approx O(n^{2.8074})$. In practice, Strassen's method has larger constant factors and can introduce more overhead than the standard algorithm for smaller matrices, but it remains an important illustration of how clever partitioning can speed up large matrix multiplications.

Task:

1. Implement Standard (Naïve) Matrix Multiplication
   (a) Write a function **naiveMultiply(A, B)** that takes two $n \times n$ matrices A and B and returns their product C.
   (b) Use the straightforward triple-nested loop approach, which runs in $O(n^3)$ time.
2. Implement Strassen's Matrix Multiplication
   (a) Write a function **strassenMultiply(A, B)** that recursively applies Strassen's algorithm.
   (b) Partition each $n \times n$ matrix into four $n/2 \times n/2$ submatrices.
   (c) Compute the seven products according to Strassen's formula.
   (d) Use these products to construct the resulting $n/2 \times n/2$ blocks of the product matrix.
   (e) Recursively apply Strassen's method until the subproblem size falls below a certain threshold (e.g., 2×2), at which point you may switch to the naïve multiplication for simplicity.

Compare Performance

- Generate random square matrices A and B of sizes $n = 64, 128, 256, 512$.

- For each n:
  (a) Record the runtime of naiveMultiply(A, B).
  (b) Record the runtime of strassenMultiply(A, B).
  (c) Present your findings in a table with four columns:
    - Size: the value of n
    - Naïve time (s): running time of Naïve method
    - Strassen time (s): running time of Strassen method
    - Correct?: whether two output matrices are equal or not

Note: Strassen's algorithm may not show advantage in running time for matrices with size of testing. Beside showing the table. Please also **upload your source code file** to Canvas.