Problem Set 3 – Module 3

shenyu li

September 23 2024

2.3.a Solving the Recurrence $T(n) = 3T(n/2)+O(n)$

given the recurrence relation:

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

assume that $O(n) \leq cn$, where c is a constant. Therefore, the recurrence becomes:

$$T(n) \leq 3T(n/2) + cn$$

expanding this recurrence step by step:

$$T(n) \leq 3\left[3T(n/4) + cn/2\right] + cn = 9T(n/4) + (1 + 3/2)cn$$

$$T(n) \leq 9\left[3T(n/8) + cn/4\right] + (1 + 3/2)cn = 27T(n/8) + \left(1 + 3/2 + (3/2)^2\right)cn$$

$$T(n) \leq 3^k T(n/2^k) + \sum_{i=0}^{k-1} \left(\frac{3}{2}\right)^i cn$$

substitute $k = \log_2 n$ into the equation, which corresponds to the point where $T(n/2^k) = T(1)$, and since $T(1) = O(1)$, get:

$$T(n) \leq 3^{\log_2 n} \cdot 1 + cn \sum_{i=0}^{\log_2 n - 1} \left(\frac{3}{2}\right)^i$$

$3^{\log_2 n} = n^{\log_2 3}$, and the sum of a geometric series $\sum_{i=0}^{k-1} r^i$ converges to:

$$\sum_{i=0}^{k-1} r^i = \frac{r^k - 1}{r - 1}$$

$$T(n) \leq n^{\log_2 3} + cn \cdot 2\left(\left(\frac{3}{2}\right)^{\log_2 n} - 1\right)$$

$$T(n) \leq n^{\log_2 3} + O(n)$$

Therefore, the time complexity of the recurrence relation is dominated by the term $n^{\log_2 3}$, and the final solution is:

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

Since $\log_2 3 \approx 1.585$, the time complexity is approximately $O(n^{1.585})$.

2.3.b Solving the Recurrence $T(n) = T(n-1)+O(1)$

given the recurrence relation:

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

assume that $O(1) \leq c$ for some constant c, so the recurrence becomes:

$$T(n) \leq T(n-1) + c$$

$$T(n) \leq T(n-2) + c + c = T(n-2) + 2c$$

$$T(n) \leq T(n-3) + c + c + c = T(n-3) + 3c$$

after k steps,

$$T(n) \leq T(n-k) + kc$$

let $k = n - 1$:

$$T(n) \leq T(1) + (n-1)c$$

Since $T(1) = O(1)$, substitute $T(1) \leq d$ for some constant d, and the recurrence simplifies to:

$$T(n) \leq d + (n-1)c$$

the time complexity of the recurrence is:

$$T(n) = O(n)$$

the final solution is:

$$T(n) = O(n)$$

2.5

Recurrence Solutions

(a) $T(n) = 2T(n/3) + 1$

apply the master theorem with $a = 2$, $b = 3$, and $f(n) = O(1)$. compute $\log_b a = \log_3 2 \approx 0.631$. Since $f(n) = O(n^0)$, which grows slower than $n^{\log_b a}$, this falls under case 1 of the master theorem. Thus, the time complexity is:

$$T(n) = O(n^{\log_3 2}) \approx O(n^{0.631})$$

(b) $T(n) = 5T(n/4) + n$

Here, $a = 5$, $b = 4$, and $f(n) = O(n)$. compute $\log_b a = \log_4 5 \approx 1.161$. Since $f(n) = O(n)$ matches $n^{\log_b a}$, this is case 2 of the master theorem. Thus, the time complexity is:

$$T(n) = O(n^{\log_4 5}) \approx O(n^{1.161})$$

(c) $T(n) = 7T(n/7) + n$

In this case, $a = 7$, $b = 7$, and $f(n) = O(n)$. Since $\log_b a = \log_7 7 = 1$ and $f(n) = O(n)$, this fits case 2 of the master theorem. Thus, the time complexity is:

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

(d) $T(n) = 9T(n/3) + n^2$

Here, $a = 9$, $b = 3$, and $f(n) = O(n^2)$. compute $\log_b a = \log_3 9 = 2$. Since $f(n) = O(n^2)$ matches $n^{\log_b a}$, this fits case 2 of the master theorem. Thus, the time complexity is:

$$T(n) = O(n^2 \log n)$$

(e) $T(n) = 8T(n/2) + n^3$

Here, $a = 8$, $b = 2$, and $f(n) = O(n^3)$. compute $\log_b a = \log_2 8 = 3$. Since $f(n) = O(n^3)$ matches $n^{\log_b a}$, this fits case 2 of the master theorem. Thus, the time complexity is:

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

(f) $T(n) = 49T(n/25) + n^{3/2} \log n$

Here, $a = 49$, $b = 25$, and $f(n) = O(n^{3/2} \log n)$. compute $\log_b a = \log_{25} 49 \approx 1.145$. Since $f(n) = O(n^{3/2} \log n)$ grows faster than $n^{\log_b a}$, this fits case 3 of the master theorem. Thus, the time complexity is:

$$T(n) = O(n^{3/2} \log n)$$

(g) $T(n) = T(n-1) + 2$

This is a linear recurrence, which expands as:

$$T(n) = T(1) + 2(n-1) = O(n)$$

Thus, the time complexity is:

$$T(n) = O(n)$$

(h) $T(n) = T(n-1) + n^c$**, where** $c \geq 1$

linear recurrence with a higher-order term n^c. Expanding the recurrence gives:

$$T(n) = T(1) + \sum_{k=1}^{n} k^c$$

The sum $\sum_{k=1}^{n} k^c$ has a time complexity of $O(n^{c+1})$. Thus, the time complexity is:

$$T(n) = O(n^{c+1})$$

2.22

The idea is to use a divide-and-conquer approach by comparing the middle elements of both arrays, and based on the comparison, discard half of one array, and continue searching in the remaining portion.

Algorithm Steps

1. If $A[m/2] \geq B[n/2]$, the elements from $A[m/2+1]$ onwards must be greater than the $(m+n)/2$ smallest elements in the combined array $A \cup B$, so discard this part of A. 2. If the current k is smaller than or equal to $(m+n)/2$, discard the elements $B[1, ..., n/2]$. 3. This process continues recursively until the base case is reached, where one of the arrays is empty, or $k = 1$, at which point the answer can be returned directly.

The recurrence relation for this approach is:

$$T(m,n) = T(m/2,n) + O(1) \quad \text{or} \quad T(m,n) = T(m,n/2) + O(1)$$

Thus, the overall time complexity is:

$$T(m,n) = O(\log m + \log n)$$

Python Code Implementation

The following Python code implements the algorithm based on the above approach:

```
def find_kth(A, B, k):
    m, n = len(A), len(B)

    # Base case: If one array is empty
    if m == 0:
        return B[k - 1]
    if n == 0:
        return A[k - 1]
```

```
    if k == 1:
        return min(A[0], B[0])

    # Find the mid points
    midA, midB = m // 2, n // 2

    # Compare A[midA] and B[midB]
    if A[midA] <= B[midB]:
        if midA + midB + 1 < k:
            # Remove the first half of A and search for k - midA - 1
            return find_kth(A[midA + 1:], B, k - midA - 1)
        else:
            # Remove the second half of B and search for k in A and B[:midB]
            return find_kth(A, B[:midB], k)
    else:
        if midA + midB + 1 < k:
            # Remove the first half of B and search for k - midB - 1
            return find_kth(A, B[midB + 1:], k - midB - 1)
        else:
            # Remove the second half of A and search for k in A[:midA] and B
            return find_kth(A[:midA], B, k)

# Example
A = [1, 3, 5, 7]
B = [2, 4, 6, 8, 9]
k = 5
print(find_kth(A, B, k))  # Output: 5
```

This algorithm computes the kth smallest element in the union of two sorted arrays in $O(\log m + \log n)$ time, as required.