Recurrence Relations

Prepared for DSC 314 course at IISER-TVM by Dr. Dhanyamol Antony.

Many algorithms are recursive in nature, and thus their time complexity is best expressed using recurrence relations. A **recurrence relation** is a mathematical model that captures the time complexity of a recursive algorithm. Solutions to recurrence relations yield the time complexity of the underlying algorithms. The methods for solving recurrence relations are explained below.

I. Substitution Method

Let $T(n)$ be the worst-case time complexity of an algorithm on input size n. We assume base cases where needed and substitute back to expand $T(n)$ iteratively.

1: Linear Search

Input: An array A of size n and an element x
Question: Is $x \in A$?

The idea behind linear search is to check whether the given element x exists in the array by scanning each element sequentially.

A recursive approach to linear search checks whether x is present in the first position. If not, it recursively searches in the rest of the array, i.e., the array excluding the first element. This reduces the problem size by one in each recursive call. Let $T(n)$ be the number of comparisons (i.e., the time) required for linear search on an array of size n.

Base case: When $n = 1$, we need one comparison:

$$T(1) = 1$$

Recurrence: For $n > 1$,

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

Expanding the recurrence:

$$T(n) = 1 + T(n-1) = 1 + 1 + T(n-2) = \cdots = (n-1) + T(1)$$

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

Thus,

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

2: Binary Search

Input: A sorted array A of size n and an element x to be searched
Question: Is $x \in A$?

Here the idea is to check whether $A[\lfloor n/2 \rfloor] = x$. If $x > A[\lfloor n/2 \rfloor]$, prune the lower half of the array $A[1, \ldots, \lfloor n/2 \rfloor]$. Otherwise, prune the upper half. Therefore, pruning happens at every iteration, reducing the problem size by half.

Let $T(n)$ be the time required for binary search on an array of size n:

$$T(n) = T(n/2) + 1, \quad T(1) = 1$$

Unrolling the recurrence:

$$T(n) = T(n/2^k) + 1 + \cdots + 1 = T(1) + k$$

When $n = 2^k$, we have $k = \log_2 n$, so:

$$T(n) = 1 + \log_2 n$$

We know that:

$$\log_2 n \leq 1 + \log_2 n \leq 2 \log_2 n, \quad \forall n \geq 2$$

Hence,

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

3: Ternary Search: In this method, we compare x with $A[n/3]$ and $A[2n/3]$. Depending on the comparisons, the problem size reduces to $n/3$.

Recurrence:

$$T(n) = T(n/3) + 2, \quad T(2) = 2$$

Unrolling:

$$T(n) = T(n/3) + 2 = T(n/9) + 2 + 2 = \cdots = T(1) + 2 \log_3 n$$
$$\Rightarrow T(n) = \Theta(\log_3 n)$$

4: k-way Search: The array is divided into k parts and compared with $k - 1$ pivots. The recurrence becomes:

$$T(n) = T(n/k) + (k - 1), \quad T(k - 1) = k - 1$$

Solving this:

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

Note: Binary, ternary, and k-way search all achieve logarithmic time complexity, differing only by the base of the logarithm:

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

5: Sorting based on Maximum Element

Here the objective to sort an array of n elements in ascending order by using a subroutine find-max which returns maximum element as a black box. To do this repeatedly find the maximum element and remove it from the array. The order in which the maximum elements are extracted is the sorted sequence. The recurrence for the above algorithm is,

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

We can further expand it:

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

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

Therefore,

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

6 Merge Sort

Divide the array into two equal subarrays and sort each subarray recursively. Continue dividing recursively until each subproblem is of size one. At this point, the problem is trivially sorted. When the recursion bottoms out, two subproblems of size one are combined to form a sorted sequence of size two. Then, two sorted subarrays of size two are merged to form a sorted array of size four, and so on. To combine two sorted arrays of size $\frac{n}{2}$ each, the worst-case number of comparisons is:

$$\frac{n}{2} + \frac{n}{2} - 1 = n - 1$$

Thus the recurrence is:

$$T(n) = 2T\left(\frac{n}{2}\right) + n - 1$$

Expanding:

$$T(n) = 2T\left(\frac{n}{2}\right) + n - 1 = 2\left[2T\left(\frac{n}{4}\right) + \frac{n}{2} - 1\right] + n - 1 = 4T\left(\frac{n}{4}\right) + n - 2 + n - 1$$

Continuing recursively:

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

When $n = 2^k$, $\frac{n}{2^k} = 1$, and $T(1) = 0$, so:

$$T(n) = 2^k \cdot 0 + \sum_{i=0}^{k-1}(n - 2^i) = \sum_{i=0}^{k-1}(n - 2^i) = kn - \sum_{i=0}^{k-1} 2^i$$

We know:

$$\sum_{i=0}^{k-1} 2^i = 2^k - 1 = n - 1$$

and $k = \log_2 n$, so:

$$T(n) = n \log_2 n - (n - 1) = n \log_2 n - n + 1$$

Therefore:

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

7: Heap Sort

This sorting algorithm is based on the data structure *max-heap*, whose construction takes $O(n)$ time which we already studied previously.

Now, let us assume that a max-heap is already provided. To sort an array, the approach is to repeatedly delete the maximum element (which is always at the root of the heap) and restore the max-heap property. Maintaining the max-heap property after deletion incurs $O(\log n)$ time per operation, since the height of a max-heap with n elements is $\log_2 n$. The order in which elements are deleted forms the sorted sequence. Let $T(n)$ denote the total number of comparisons required to sort an array of size n using heap sort. Then the recurrence is:

$$T(n) = T(n - 1) + \log_2 n, \quad T(1) = 0$$

Expanding the recurrence:

$$T(n) = T(n - 2) + \log_2(n - 1) + \log_2(n) = T(1) + \sum_{i=2}^{n} \log_2 i$$

This is equivalent to:

$$T(n) = \log_2(2 \cdot 3 \cdot \ldots \cdot n) = \log_2(n!)$$

Using Stirling's approximation:

$$\log_2(n!) \leq n \log_2 n \quad \text{and} \quad \log_2(n!) = \Omega(n \log_2 n)$$

Therefore:

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

8: Finding Maximum

To find the maximum in an array of n elements, we assume the first element is the current maximum and compare it linearly with the rest of the elements. If a larger element is found, we update the current maximum.

Let $T(n)$ denote the worst-case number of comparisons to find the maximum element in an array of size n.

Base Case:

$$T(1) = 0 \quad \text{or} \quad T(2) = 1$$

Recurrence Relation:

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

Solving the recurrence:

$$T(n) = T(n-1) + 1 = T(n-2) + 2 = \cdots = T(1) + (n-1) = 0 + (n-1)$$

Therefore:

$$T(n) = n - 1 = \Theta(n)$$

II. Change of Variable Technique

1: $T(n) = 2T(\sqrt{n}) + 1$**,** $T(1) = 1$

Let $n = 2^m$. Then:

$$T(2^m) = 2T(\sqrt{2^m}) + 1 = 2T(2^{m/2}) + 1$$

Define $S(m) = T(2^m)$, then:

$$S(m) = 2S(m/2) + 1$$

Unrolling gives:

$$S(m) = 2(2S(m/4) + 1) + 1 = 2^2 S(m/4) + 2 + 1 = \cdots$$

$$= 2^k S(m/2^k) + 2^{k-1} + \cdots + 1$$

Assume $m = 2^k \Rightarrow S(m/2^k) = S(1) = T(2) \approx 3$, since T counts comparisons and must be an integer.

Therefore:

$$S(m) = 3 + (2^k - 1) = m + 2$$

Since $S(m) = T(2^m)$, we get:

$$T(n) = \log_2 n + 2 = \Theta(\log n)$$

2: $T(n) = 2T(\sqrt{n}) + n$**,** $T(1) = 1$

Let $n = 2^m \Rightarrow T(2^m) = 2T(2^{m/2}) + 2^m$

Define $S(m) = T(2^m)$, then:

$$S(m) = 2S(m/2) + 2^m$$

Expand the recurrence:

$$\begin{aligned} S(m) &= 2S(m/2) + 2^m \\ &= 2\left[2S(m/4) + 2^{m/2}\right] + 2^m = 4S(m/4) + 2^{m/2+1} + 2^m \\ &= 8S(m/8) + 2^{m/4+2} + 2^{m/2+1} + 2^m \\ &\vdots \\ &= 2^k S(m/2^k) + \sum_{i=0}^{k-1} 2^{m/2^i+i} \end{aligned}$$

Stop when $m/2^k = 1 \Rightarrow k = \log_2 m$, and $S(1) = c$, so:

$$S(m) = m \cdot c + \sum_{i=0}^{\log m - 1} 2^{m/2^i+i}$$

Now each term in the sum is of the form:

$$2^{m/2^i+i} = 2^{m/2^i} \cdot 2^i$$

The dominant term is when $i = 0$, i.e., 2^m. All other terms are smaller. Since there are at most $\log m$ terms, the total sum is:

$$\sum_{i=0}^{\log m - 1} 2^{m/2^i+i} = O(2^m)$$

and since from the first term $S(m) \geq 2^m$, we get:

$$T(n) = \Omega(n)$$

Thus:

$$S(m) = \Theta(2^m) \Rightarrow T(2^m) = S(m) = \Theta(2^m) \Rightarrow T(n) = \Theta(n)$$

$\Rightarrow$

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

3: $T(n) = 2T(\sqrt{n}) + \log n$**,** $T(1) = 1$

Let $n = 2^m \Rightarrow T(2^m) = 2T(2^{m/2}) + m$

Define $S(m) = T(2^m)$, so:

$$S(m) = 2S(m/2) + m$$

Unrolling:

$$S(m) = 2(2S(m/4) + m/2) + m = 2^2 S(m/4) + m + m = \cdots$$
$$= 2^k S(m/2^k) + k \cdot m$$

Assume $m = 2^k \Rightarrow S(1) = T(2) \approx 2$, so:

$$S(m) = 2 + m \log m \Rightarrow S(m) = O(m \log m)$$

Therefore:

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

III. Master Theorem method

We shall now consider the *Master Theorem*, which serves as a "cookbook" for many well-known recurrence relations. It presents a framework and formulae using which solutions to many recurrence relations can be obtained very easily. Almost all recurrences of the type

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

can be solved easily by doing a simple check and identifying one of the three cases mentioned in the following theorem.

By comparing $n^{\log_b a}$ (which corresponds to the total contribution from the leaves of the recursion tree) with $f(n)$ (the work done at each level), one can decide upon the time complexity of the algorithm.

Master Theorem: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a non-negative function, and let $T(n)$ be defined on the non-negative integers by the recurrence

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n),$$

where we interpret n/b to be either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

Case 1: If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then

$$T(n) = \Theta(n^{\log_b a}).$$

Case 2: If $f(n) = \Theta(n^{\log_b a})$, then

$$T(n) = \Theta(n^{\log_b a} \log n).$$

Case 3: If $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon > 0$, and if there exists a constant $c < 1$ and sufficiently large n such that

$$a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n),$$

then

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

Examples using Master Theorem

1.$T(n) = 9T(\frac{n}{3}) + n$

$\Rightarrow$

$$a = 9, \quad b = 3, \quad f(n) = n$$

Then,

$$n^{\log_b a} = n^{\log_3 9} = n^2$$

Comparing $f(n) = n$ with $n^{\log_b a} = n^2$, we see that:

$$f(n) = O(n^{2-\varepsilon}) \quad \text{for } \varepsilon = 1$$

This satisfies **Case 1** of the Master Theorem. Hence,

$$T(n) = \Theta(n^{\log_3 9}) = \Theta(n^2)$$

2. $T(n) = T\left(\frac{2n}{3}\right) + 1$

We identify:

$$a = 1, \quad b = \frac{3}{2}, \quad f(n) = 1$$

Then,

$$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$$

Comparing $f(n) = 1$ with $n^{\log_b a} = 1$, we see that:

$$f(n) = \Theta(n^{\log_b a})$$

This satisfies **Case 2** of the Master Theorem. Hence,

$$T(n) = \Theta(n^{\log_{3/2} 1} \log n) = \Theta(\log n)$$

3. $T(n) = 2T\left(\frac{n}{2}\right) + n$

$$a = 2, \quad b = 2, \quad f(n) = n$$
$$n^{\log_b a} = n^{\log_2 2} = n \Rightarrow f(n) = \Theta(n^{\log_b a})$$

Case 2 of Master Theorem applies.

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

4. $T(n) = 3T\left(\frac{n}{4}\right) + n \log n$

$$a = 3, \quad b = 4, \quad f(n) = n \log n, \quad n^{\log_b a} = n^{\log_4 3}$$

Since $f(n) = \Omega(n^{\log_4 3 + \varepsilon})$ for some $\varepsilon > 0$, we check the regularity condition:

$$af(n/b) = 3 \cdot \frac{n}{4} \log\left(\frac{n}{4}\right) \leq c \cdot n \log n \text{ for } c < 1$$

Case 3 applies.

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

5. $T(n) = 4T\left(\frac{n}{2}\right) + n$

$$a = 4, \quad b = 2, \quad f(n) = n, \quad n^{\log_2 4} = n^2$$

$$f(n) = O(n^{2-\varepsilon}) \Rightarrow \textbf{Case 1 applies}$$
$$T(n) = \Theta(n^2)$$

6. $T(n) = 4T\left(\frac{n}{2}\right) + n^2$

$$a = 4, \quad b = 2, \quad f(n) = n^2, \quad n^{\log_2 4} = n^2$$

$$f(n) = \Theta(n^{\log_2 4}) \Rightarrow \textbf{Case 2 applies}$$
$$T(n) = \Theta(n^2 \log n)$$

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

$$a = 2, \quad b = 2, \quad f(n) = 2^n, \quad n^{\log_2 2} = n$$

Here,

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

We compute:

$$n^{\log_b a} = n^{\log_2 2} = n$$

So,

$$f(n) = 2^n = \Omega\left(n^{\log_2 2+\varepsilon}\right) = \Omega(n^{1+\varepsilon}) \quad \text{for any } \varepsilon > 0$$

Now we check the regularity condition:

$$a \cdot f\left(\frac{n}{b}\right) = 2 \cdot 2^{n/2} = 2^{n/2+1} \leq c \cdot 2^n$$

$$\text{For large } n, \text{ e.g., } n \geq 4, \text{ we can choose } c = \frac{1}{2} < 1$$

Hence, the regularity condition holds, and by **Case 3** of the Master Theorem:

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

8. $T(n) = 2T\left(\frac{n}{2}\right) + n3^n$

$$a = 2, \quad b = 2, \quad f(n) = n3^n$$

$$n^{\log_b a} = n^{\log_2 2} = n$$

Compare $f(n) = n3^n$ with $n^{\log_b a} = n$. Clearly,

$$f(n) = \Omega\left(n^{1+\varepsilon}\right) \text{ for } \varepsilon > 0$$

Check the Regularity condition:

$$a \cdot f\left(\frac{n}{b}\right) = 2 \cdot \left(\frac{n}{2}\right)^3 = 2 \cdot \frac{n^3}{8} = \frac{n^3}{4}$$

We want:

$$af(n/b) \leq c \cdot f(n)$$

$$\frac{n^3}{4} \leq c \cdot n^3 \Rightarrow c \geq \frac{1}{4}$$

So, the regularity condition is satisfied for $c = \frac{1}{3} < 1$ and all $n \geq 1$.

Hence, by Case 3 of the Master Theorem,

$$T(n) = \Theta(f(n)) = \Theta(n3^n)$$

Characteristic Equation Method

Used for solving linear recurrences like:

$$T(n) = aT(n-1) + bT(n-2)$$

The general form of solution:

$$T(n) = \alpha r_1^n + \beta r_2^n$$

where r_1, r_2 are roots of the characteristic equation.

Example: Fibonacci

$$T(n) = T(n-1) + T(n-2) \Rightarrow r^2 = r + 1 \Rightarrow r = \frac{1 \pm \sqrt{5}}{2}$$