

Recurrence Relations

Prem Nair

Binary Search

Algorithm binSearch(A, x, lower, upper)

Input: Already sorted array A of size n, value x to be
searched for in array section A[lower]..A[upper]

Output: true or false

if lower > upper **then return** false

mid \leftarrow (upper + lower)/2

if x = A[mid] **then return** true

if x < A[mid] **then**

return binSearch(A, x, lower, mid - 1)

else

return binSearch(A, x, mid + 1, upper)

For the worst case (x is above all elements of A and n a power of 2), running time is given by
the **Recurrence Relation:** (In this case, right half is always half the size of the original.)

$$T(1) = d; \quad T(n) = c + T(n/2)$$

Binary Search

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

$$a = 1, b = 2, k = 0. a = b^0$$

Apply Master Theorem

$$n^k \log_b n = \log_2 n.$$

Hence by Master Theorem, we have $\Theta(\log_2 n)$

Binary Search

Visualization

There is a depth of $\Theta(\log n)$.

Width = $\Theta(1)$.

Hence time complexity is $\Theta(\log n)$.

FindMax(A, lower, upper)

Algorithm FindMax(A, lower, upper)

Input: An unsorted array A[lower, ..., upper]

Output: max value

if lower = upper **then return** A[lower]

mid \leftarrow (upper + lower)/2

left \leftarrow FindMax(A, lower, mid)

right \leftarrow FindMax(A, mid + 1, upper)

return max(left, right)

For the worst case (x is above all elements of A and n a power of 2), running time is given by the **Recurrence Relation**: (In this case, right half is always half the size of the original.)

$$T(1) = d; \quad T(n) = c + 2T(n/2)$$

$$T(1) = d, T(n) = 2T(n/2) + c \quad (n > 1)$$
$$a = 2, b = 2, k = 0. a > b^0$$

Apply Master Theorem

$$\log_b a = \log_2 2 = 1.$$

Hence by Master Theorem, we have $\Theta(n)$

FindMax(A, lower, upper)

Visualization

There is a depth of $\Theta(\log n)$.

Width is not a constant. It is 1, 2, ..., $n/2$.

$$1 + 2 + \dots + n/2 = n - 1$$

Hence the time complexity is $\Theta(n)$.

FindMax2(A, lower, upper)

Algorithm FindMax2(A, lower, upper)

Input: An unsorted array A[lower, ..., upper]

Output: max value

if lower = upper **then return** A[lower]

left \leftarrow A[lower]

right \leftarrow FindMax2(A, lower + 1, upper)

return max(left, right)

For the worst case (x is above all elements of A and n a power of 2), running time is given by the ***Recurrence Relation***

$$T(1) = d; \quad T(n) = c + T(n-1)$$

FindMax2

You cannot apply master theorem.

$$T(n) = T(n - 1) + c = T(n - 2) + 2c = \dots = T(1) + (n - 1)c = d + (n - 1)c \text{ is } \Theta(n).$$

Visualization

There is a depth of $\Theta(n)$.

Width is 1.

Hence time complexity is $\Theta(n)$

Merge Sort

Algorithm MergeSort(A, lower, upper)

Input: An unsorted array A[lower, ..., upper]

Output: Sorted array

if lower = upper **then return** A[lower]

mid \leftarrow (upper + lower)/2

L \leftarrow MergeSort(A, lower, mid)

R \leftarrow MergeSort (A, mid + 1, upper)

return merge(L, R)

For the worst case (x is above all elements of A and n a power of 2), running time is given by the **Recurrence Relation:** (In this case, right half is always half the size of the original.)

$$T(1) = d; \quad T(n) = 2T(n/2) + cn \quad // \text{ n for merge}$$

$$T(1) = d, T(n) = 2T(n/2) + cn \quad (n > 1)$$
$$a = 2, b = 2, k = 1. a = b^1$$

Apply Master Theorem

$$k = 1$$

Hence by Master Theorem, we have $\Theta(n^k \log n)$. That is, $\Theta(n \log n)$.

Visualization

There is a depth of $\Theta(\log n)$

Width = $\Theta(n)$.

Hence time complexity is $\Theta(n \log n)$.

The Master Theorem

For recurrences that arise from Divide-And-Conquer algorithms (like Binary Search), there is a general formula that can be used.

Theorem. Suppose $T(n)$ satisfies

$$T(n) = \begin{cases} d & \text{if } n = 1 \\ aT(\lceil \frac{n}{b} \rceil) + cn^k & \text{otherwise} \end{cases}$$

where k is a nonnegative integer and a, b, c, d are constants with $a > 0, b > 1, c > 0, d \geq 0$. Then

$$T(n) = \begin{cases} \Theta(n^k) & \text{if } a < b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^{\log_b a}) & \text{if } a > b^k \end{cases}$$