Recurrence Relations

Prem Nair

$T(1) = d,\ T(n) = aT(n/b) + cn^k \quad (n > 1)$
Assume $n = b^p$ or $p = \log_b n$

$T(b^p) = aT(b^{p-1}) + c(b^p)^k$

$aT(b^{p-1}) = a^2T(b^{p-2}) + ac(b^{p-1})^k$

$a^2T(b^{p-2}) = a^3T(b^{p-3}) + a^2c(b^{p-2})^k$

. . .

$a^{p-2}T(b^2) = a^{p-1}T(b) + a^{p-2}c(b^2)^k$

$a^{p-1}T(b) = a^pT(1) + a^{p-1}c(b)^k$

$T(n) = c(b^p)^k + ac(b^{p-1})^k + a^2c(b^{p-2})^k + \ldots + a^{p-2}c(b^2)^k + a^{p-1}c(b)^k + a^pT(1)$

$T(n) = c[\,(b^p)^k + a(b^{p-1})^k + a^2(b^{p-2})^k + \ldots + a^{p-2}(b^2)^k + a^{p-1}(b)^k\,] + a^pd.$

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 ← (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.)

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

Binary Search

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

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

$T(n) = c[\,(b^p)^k + a(b^{p-1})^k + a^2(b^{p-2})^k + \ldots + a^{p-2}(b^2)^k + a^{p-1}(b)^k\,] + a^p d.$

$T(n) = c[\,1 + 1 + 1 + \ldots + 1 + 1] + d = cp + d = O(p) = O(\log n)$

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; T(n) = c + 2T(n/2)

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

$T(n) = c[(b^p)^k + a(b^{p-1})^k + a^2(b^{p-2})^k + . . . + a^{p-2}(b^2)^k + a^{p-1}(b)^k]+ a^p d.$

$T(n) = c[1 + 2 + 2^2 + . . . + 2^{p-2} + 2^{p-1}]+ 2^p d = O(2^p) = O(n)$

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.

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

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; T(n) = c + T(n-1)

FindMax2

You cannot apply master theorem.

$T(n) = T(n - 1) + c = T(n - 2) + 2c = \ldots = T(1) + (n - 1)c = d + (n-1)c$ 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; **T(n) = 2T(n/2) +cn // n for merge**

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

$T(n) = c[\ (b^p)^k + a(b^{p-1})^k + a^2(b^{p-2})^k + \ldots + a^{p-2}(b^2)^k + a^{p-1}(b)^k\ ] + a^p d.$

$T(n) = c[\ (2^p) + 2(2^{p-1}) + 2^2(2^{p-2}) + \ldots + 2^{p-2}(2^2) + 2^{p-1}(2)\ ] + d < O(n\log n)$

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 Θ(log n)

Width = Θ(n).

Hence time complexity is Θ(nlog 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

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