Name : Shraddha Jamadade
Student Id : 110287963

classmate
Date
Page

CSci 174 - Assignment 2

Master Method

- Master Method is a powerful tool for solving recurrences.
- It is a 'black box' for solving recurrences.
- Assumption : All subproblems have equal size.
- Recurrences have 2 ingredients :

1) When the input size drops to a sufficiently small size, the recursion is stopped and the problem is solved in constant time.

   Base Case : $T(N) = a$ constant

2) For all larger N :

$$\boxed{T(N) \leq a\,T(N/b) + O(N^d)}$$

where,

a = number of recursive calls (≥ 1)

b = input size shrinkage factor (>1)

d = amount of work done outside recursive call (exponent in running time of combine step (≥ 0))

a, b, d are constants independent of N

The running time of inputs of size N is over bounded by one of 3 things

3 Cases

$$T(N) = \begin{cases} O(N^d \log N) & \text{if } a = b^d \quad (\text{Case } 1) \\ O(N^d) & \text{if } a < b^d \quad (\text{Case } 2) \\ O(N^{\log_b a}) & \text{if } a > b^d \quad (\text{Case } 3) \end{cases}$$

The trigger which determines which case you are in is a comparison between two numbers : a & b^d

Date
Page

(1) Use the Master Method to solve the recurrences below

(a) $T(N) = 7 * T(N/7) + N$

$a = 7$
$b = 7$
$d = 1$

Comparing the two values a and b^d

Case I : $a = b^d$
$\rightarrow 7 = 7^1$
$\rightarrow 7 = 7$

Thus case 1 is true.
In case 1, the running time is bounded by

$T(N) \leq O(N^d \log N)$
$T(N) = O(N \log N)$ .... $d = 1$

(b) $T(N) = 7 * T(N/27) + N$

Comparing a and b^d, Here $a = 7$, $b = 27$, $d = 1$

Case 1 : $a = b^d$
$\rightarrow 7 = 27^1$
$\rightarrow 7 = 27$

Case 1 is false.

Case 2 : $a < b^d$

$\rightarrow 7 < 27^1$

$\rightarrow 7 < 27$

Case 2 is true.

In case 2, the running time is bounded by

$T(N) = O(N^d)$

$T(N) = O(N^1)$

$T(N) = O(N)$

(c) $T(N) = 25 * T(N/2) + N^3$

Here, $a = 25$

$b = 2$

$d = 3$

Comparing a and b^d,

Case 1 : $a = b^d$

$\rightarrow 25 = 2^3$

$\rightarrow 25 = 8$

Case 1 is false.

Case 2 : $a < b^d$

$\rightarrow 25 < 2^3$

$\rightarrow 25 < 8$

Case 2 is false.

Case 3 : $a > b^d$

$\rightarrow 25 > 2^3$

$\rightarrow 25 > 8$

Case 3 is true

In case 3, the running time is bounded by

$$T(N) = O(N^{\log_b a})$$

$$\boxed{T(N) = O(N^{\log_2 25})}$$

or

$$T(N) = O\left(N^{\left(\frac{\log 25}{\log 2}\right)}\right)$$

$$T(N) = O(N^{4.64})$$

$$T(N) = O(N^{\log_2 5^2})$$

$$T(N) = O(N^{2 * \log_2 5})$$

(2) Consider the following pseudo-code :

```
MAX-HEAPIFY (A, i)
  l = LEFT(i)
  r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
  largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
  largest = r
if largest ≠ i
exchange A[i] with A[largest]
MAX-HEAPIFY (A, largest)
```

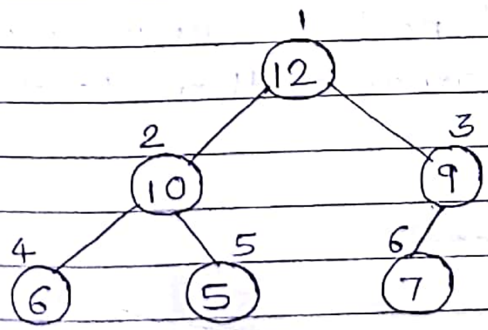
With the utility functions :

PARENT(i)
1 return $\lfloor i/2 \rfloor$

LEFT(i)
1 return 2i

RIGHT(i)
1 return 2i+1

- The heap data structure is an array object that can be viewed as a nearly complete binary tree
- Each node of the tree corresponds to an element of the array that stores the value in the node.



1	2	3	4	5	6
12	10	9	6	5	7

- The number within the circle at each node in the tree is the value stored at that node.
- The number above the node is the corresponding index in the array
- In the array, parents are always to the left of their children.

The tree has height two.
The node at index 2 (value 10) has height 1.

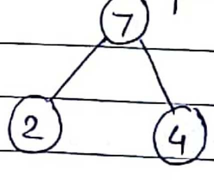
- The root of the tree is A[1]
- Given the index i of a node, the indices of its parent - PARENT(i), left child - LEFT(i), right child - RIGHT(i) is computed as shown in the utility functions above

- In max-heap, for every node i other than root,
  A[PARENT(i)] ≥ A[i],
  that is the value of a node is at the most the value of its parent.
- The largest element in a max heap is stored at the root.

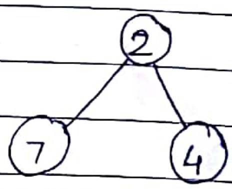
- The MAX-HEAPIFY procedure, runs in O(log n) time

- In this function, the parent node is swapped with the largest child node recursively until the parent node is greater than its child nodes.

- Example :

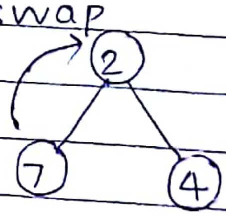


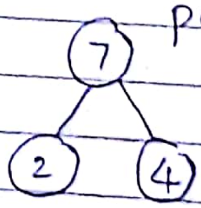
parent is already larger.



child is larger than parent

swap





After swapping parent is largest

- The running time of MAX-HEAPIFY on a subtree of size n rooted at given node i, is $\Theta(1)$
- The children subtrees have size at most $2n/3$
- In the worst case, the last row of the tree is exactly half

- The running time of MAX-HEAPIFY can be described by the recurrence :

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

$$T(n) = T\left(\frac{2}{3}N\right) + \Theta(N^0)$$

For a level i, no. of nodes are 2^i
The last level k is half, $\therefore$ no. of nodes: $\frac{2^k}{2}$

For the first $k-1$ levels, no. of nodes is

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

Thus total nodes are $n = 2^k - 1 + \frac{2^k}{2}$

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

$$n = 2^k\left(\frac{3}{2}\right) - 1$$

$$n + 1 = 2^k\left(\frac{3}{2}\right)$$

$$2^k = \frac{2}{3}(n+1)$$

$\therefore \quad T(N) = T\left(\frac{2}{3}N\right) + \Theta(1)$

$T(N) = T\left(\frac{2}{3}N\right) + \Theta(N^0)$

~~Here, a=1, b=~~

$T(N) = T\left(\frac{N}{3/2}\right) + \Theta(N^0)$

Here, $a = 1$,

$b = \frac{3}{2}$

$d = 0$

Comparing the two values a and b^d

Case I : $a = b^d$

$1 = \left(\frac{3}{2}\right)^0$

$1 = 1$

Thus case 1 is true.

In case 1, the running time is bounded by

$T(N) = O(N^d \log N)$

$T(N) = O(N^0 \log N)$

$T(N) = O(\log N) \quad \ldots \quad N^0 = 1$

(3) Consider an algorithm where we take in a prob. and create 3 subproblems each half as large as the original problem.

$\therefore$ Number of subproblems = 3

$a = 3$.

Then the sub problem solutions are combined with a single loop through a list $\frac{1}{3}$th the size of the original problem.

$\therefore$ Size of each subproblem = $\frac{1}{2}$

$b = \frac{1}{2}$

There is single loop

$\therefore d = 1$

Merging requires $\frac{1}{3}N$ (N = size of orig. prob.)

$$T(N) = 3T\left(\frac{N}{2}\right) + O\left(\left(\frac{N}{3}\right)^1\right)$$

The recurrence relation is given by the above statement

$a = 3$, $b = 2$, $d = 1$

Comparing a & b^d

$3 > 2^1$

$T(N) = O(N^{\log_b a})$

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

Scanned by CamScanner

classmate

Date
Page

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

$$T(N) = O(N^{3.65})$$

References: 1) Coursera: Divide and Conquer, Sorting and Searching, and Randomized Algorithms - Tim Roughgarden
https://www.coursera.org/learn/algorithms-divide-conquer/home/week/2

2) 'Introduction to Algorithms' - Thomas H. Cormen, Charles E. Leiserson.