

# 6.006 Cheat Sheet (shreyask)

## Recurrences

The master theorem concerns recurrence relations of the form:

$$T(n) = a T\left(\frac{n}{b}\right) + f(n) \text{ where } a \geq 1, b > 1$$

In the application to the analysis of a recursive algorithm, the constants and function take on the following significance:

- $n$  is the size of the problem.
- $a$  is the number of subproblems in the recursion.
- $n/b$  is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)
- $f(n)$  is the cost of the work done outside the recursive calls, which includes the cost of dividing the problem and the cost of merging the solutions to the subproblems.

It is possible to determine an asymptotic tight bound in these three cases:

### Case 1 [edit]

**Generic form** [edit]

If  $f(n) = O(n^c)$  where  $c < \log_b a$  (using **big O notation**)

then:

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

**Example** [edit]

$$T(n) = 8T\left(\frac{n}{2}\right) + 1000n^2$$

As one can see from the formula above:

$$a = 8, b = 2, f(n) = 1000n^2, \text{ so } f(n) = O(n^c), \text{ where } c = 2$$

Next, we see if we satisfy the case 1 condition:

$$\log_b a = \log_2 8 = 3 > c.$$

It follows from the first case of the master theorem that

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$$

(indeed, the exact solution of the recurrence relation is

$$T(n) = 1001n^3 - 1000n^2, \text{ assuming } T(1) = 1).$$

### Case 2 [edit]

**Generic form** [edit]

If it is true, for some constant  $k \geq 0$ , that:

$$f(n) = \Theta(n^c \log^k n) \text{ where } c = \log_b a$$

then:

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

**Example** [edit]

$$T(n) = 2T\left(\frac{n}{2}\right) + 10n$$

As we can see in the formula above the variables get the following values:

$$a = 2, b = 2, c = 1, f(n) = 10n \\ f(n) = \Theta(n^c \log^k n) \text{ where } c = 1, k = 0$$

Next, we see if we satisfy the case 2 condition:

$$\log_b a = \log_2 2 = 1, \text{ and therefore, } c = \log_b a$$

So it follows from the second case of the master theorem:

$$T(n) = \Theta(n^{\log_b a} \log^{k+1} n) = \Theta(n^1 \log^1 n) = \Theta(n \log n)$$

Thus the given recurrence relation  $T(n)$  was in  $\Theta(n \log n)$ .

### Case 3 [edit]

**Generic form** [edit]

If it is true that:

$$f(n) = \Omega(n^c) \text{ where } c > \log_b a$$

and if it is also true that:

$$af\left(\frac{n}{b}\right) \leq kf(n) \text{ for some constant } k < 1 \text{ and sufficiently large } n \\ (\text{often called the regularity condition})$$

then:

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

**Example** [edit]

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

As we can see in the formula above the variables get the following values:

$$a = 2, b = 2, f(n) = n^2 \\ f(n) = \Omega(n^c), \text{ where } c = 2$$

Next, we see if we satisfy the case 3 condition:

$$\log_b a = \log_2 2 = 1, \text{ and therefore, yes, } c > \log_b a$$

The regularity condition also holds:

$$2\left(\frac{n^2}{4}\right) \leq kn^2, \text{ choosing } k = 1/2$$

So it follows from the third case of the master theorem:

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

## Heaps

**root of tree** is  $i = 0$

**parent(i)** =  $i/2$ , **left(i)** =  $2i$ , **right(i)** =  $2i + 1$

**max-heap prop:** the key of the node is  $\geq$  the keys of its children.

max height of  $\log n$ , almost binary tree

**max-heapify:** assumes the left and right subtrees are maxheaps. look at both children, if higher than both: stop, else exchange from the larger children. Run maxheapify again on the same element in new position.  $O(\log n)$ .

**build-heap:** start  $n/2$  and go down till the first element and call max-heapify on every element.  $O(n)$ .

**complexities:** find-max  $O(1)$ , delete-max  $O(\log n)$ , insert  $O(\log n)$ , decrease-key,  $O(\log n)$ , merge, linear.

## Binary Search Tree

Useful for searching+inserting stuff.

Each parent has access to child, each child to parent. Left sub-tree is less than parent, right sub-tree is greater.

Everything  $O(n)$  worst,  $O(\log n)$  average

**insert:** search but when reach leaf insert left or right.  $O(h)$

**find-min:** go till the left leaf or right leaf.

**in-order traversal:**

recursively iterate through left subtree, print root, recursively iterate through right subtree.

## AVL

### Description of Rotations and Stuff

Tree is balanced is  $h = O(\log n)$

**height of tree:** longest path from root to leaf

**height of node:** longest path from that node to a leaf, recursively  $h(node) = \max(h(left), h(right)) + 1$ . The null pointers at the end of the leaves are height -1.

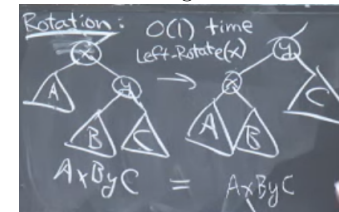
AVL Trees need the heights of left and right children of every node to differ by at most 1.

**AVL Insert:**

1. Simple BST insert

2. Fix AVL property from changed node up

Rotation changes order of nodes and stuff in constant time and maintains the BST property and the in-order order. Rotation is done at the highest unbalanced node first.



### Proof That Difference of 1 is balanced

$N_h$  is the minimum number of nodes that's possible of height  $h$ . Since the two sub trees differ by height 1,

$$\begin{aligned} N_h &= 1 + N_{h-1} + N_{h-2} \\ &> 1 + 2N_{h-2} \\ &> 2N_{h-2} \\ &= \Theta(2^{n/2}) \\ \implies h &< 2 \log n \end{aligned}$$

## Practically Useful Stuff

Everything in  $O(\log n)$  time. Access, Search, Insertion, Deletion, Find-Max, Find-Min, Successor, Predecessor and stuff. (Cause height is maintained.)

## Search Bound

**Decision Tree:** any comparison algorithm can be viewed as a tree of all possible comparisons and their outcomes and resulting answer.

For searching, tree is binary and must have at least  $n$  leaves for each answer. Height has to be at least  $\log n$

## Comparison Sort Lower Bound

Decision tree is binary, number of leaves is greater than number of possible answers, which is  $n!$ . Height  $> \log n! \implies$  Height  $> O(n \log n)$

## Cheaty Sorts

### Counting Sort


Allocate memory with the number of keys with zeros. Go through given array and update the counts in the count array. Cumulative frequencies and then place from last to first in that spot.

$O(n + k)$  where  $k$  the maximum value an element can take.

## Radix Sort

Execute stable counting sort from MSB to LSB.  $O(wN)$ .  $w$  is the number of bits in a word.

## Python Costs

Operation	Average Case	 Amortized Worst Case
Copy	$O(n)$	$O(n)$
Append[1]	$O(1)$	$O(1)$
Insert	$O(n)$	$O(n)$
Get Item	$O(1)$	$O(1)$
Set Item	$O(1)$	$O(1)$
Delete Item	$O(n)$	$O(n)$
Iteration	$O(n)$	$O(n)$
Get Slice	$O(k)$	$O(k)$
Del Slice	$O(n)$	$O(n)$
Set Slice	$O(k+n)$	$O(k+n)$
Extend[1]	$O(k)$	$O(k)$
 Sort	$O(n \log n)$	$O(n \log n)$
Multiply	$O(nk)$	$O(nk)$
x in s	$O(n)$	
min(s), max(s)	$O(n)$	
Get Length	$O(1)$	$O(1)$

## Hash Tables

Pre-Hashing, whatever key we have, we convert to non-negative integer by just taking the binary representation

of that object  $\rightarrow$  integer. **Chaining** if collision, store as a list. Worst case  $O(n)$ , any hashing. But randomized?

**SUHA:** each key is equally likely to be hashed to any slot of the table, independent of each other.

**Proof of Constant Time:** expected length of chain  $n/m = \alpha$  load factor.  $n$  is keys,  $M$  slots.

$$collisions = \frac{N(N-1)}{2} \frac{1}{M}$$

$$P(collision) = 1/m$$

$$P(query\ correct) = (1 - 1/m)^{n-1}$$

## Random Useful Stuff

$$f(n) = \sum_{x=1}^n \log x = \Theta(n \log n)$$

Only heap sort isn't stable.

Only insertion and heap is in place.

---

Copyright © 2014 Winston Chang

<http://www.stdout.org/~winston/latex/>