Q-1: [Marks 8 + 8]

1. Given $f(n) = a^n$ and $g(n) = b^n$ where the constants a and b are greater than 1, i.e. $a > 1, b > 1$. Show that if $a \neq b$ then $f(n) \neq \Theta(g(n))$.

   Solution

   Let c = a/b

   If a $\neq$ b then there are two cases

   1) a>b $\Rightarrow$ c > 1

   $$\lim_{n\to\infty} \frac{a^n}{b^n} = \lim_{n\to\infty} (\frac{a}{b})^n = \lim_{n\to\infty} c^n = \infty$$ because c > 1

   2) a<b $\Rightarrow$ c < 1

   $$\lim_{n\to\infty} \frac{a^n}{b^n} = \lim_{n\to\infty} (\frac{a}{b})^n = \lim_{n\to\infty} c^n = 0$$ because c < 1

2. Given $f(n) = \log_a n$ and $g(n) = \log_b n$ where the constants a and b are greater than 1, i.e. $a > 1, b > 1$. Show that $f(n) = \Theta(g(n))$.

   Solution

   $$\log_b n = \frac{\log_a n}{\log_a b} \Rightarrow \log_b n = c * \log_a n \text{ where } c = \frac{1}{\log_a b}$$

   Because c is positive constant (both a and b are larger than 1), hence the claim holds by definition of theta notation.

Q-2: [Marks 3 + 3 + 3] Justify your answer. No credit without justification.

1. An algorithm that uses 2^{32} units of extra space is still "in-place" [T/F].

   True; because 2^{32} is constant and an in-place algorithm is allowed to have constant extra memory.

2. A sorting algorithm is called "stable" if it does not get stuck in an infinite loop or recursion [T/F].

   False; stability is a more specific term to sorting algorithm. If a sorting algorithm preserves the original ordering of the repeated elements in the output, then the sorting algorithm is called "stable".

3. If all the elements of an array of size "n" are integers and distinct then counting sort runs in $O(n)$ [T/F].

   False; the counting sort depends upon the range of the data to be sorted. Even with distinct +ive integers, you can have very large range and the over all time may not be O(n).

Q-3: [Marks 10] Prove or disprove that any binary Max-heap (in general) can be converted into some binary search tree in $O(n)$ just by applying suitable "if" conditions and doing the required swaps at each node of the binary max-heap.

Solution

The claim is false because if a Max-heap can be converted into some binary search tree in O(n), then, One can do the in-order-traversal of the binary search tree to get the sorted elements in O(n). Because one can build Max-heap from an array in O(n) and if the Max-heap can be converted to binary search tree in O(n), then one can do sorting in O(n) ,in general, which is impossible (because sorting is $\Omega(n \lg n)$).

Q-4: [Marks 10] Find the asymptotic tight bound ($\Theta - bound$) for the following recurrence (You are free to apply any method):

$$T(n) = T(\frac{n}{7}) + T(\frac{3n}{4}) + n$$

$$T(1) = 1$$

Solution:

Use recursion-tree method to solve, T(n) = $\Theta(n)$; by the way it is the recurrence equation of k^{th} smallest algorithm if we use 7 sized columns instead of 5.

Q-5: [Marks 5+5] Assuming $T(2) = 1$, write and solve (using any method) the recurrence-equations for the following divide and conquer algorithms:

- Algorithm **A** solves the problem of size "n" by dividing it into 4 subproblems each of size n/2 (half of the total problem size), recursively solving each subproblem, and then combining the solutions in $\Theta(n^2 \lg n)$ time.
- Algorithm **B** solves the same problem of size "n" by dividing it into 9 subproblems each of size n/2 (half of the total problem size), recursively solving each subproblem, and then combining the solutions in $\Theta(\lg n)$ time.

Which algorithm is asymptotically faster?

Solution:

A) $T(n) = 4T(n/2) + \Theta(n^2 \lg n)$; master method case-2,
$T(n) = \Theta(n^2 \lg^2 n)$

B) $T(n) = 9T(n/2) + \Theta(\lg n)$; master method case-1,
$T(n) = \Theta(n^{3.something})$

So Algorithm A is faster

Q-6: [Marks 15] Given an array A of "n" distinct elements, consider an algorithm below:

```
a = foo(A,i,j) {

    if (j<i)
       return NULL
    end
    if i==j
       return A[i]
    end
    idx = getRandomPivotIndex(i,j)
    k = partition(A,i,j,idx)
    if k == j
        return A[k]
    else
        return foo(A,k+1,j)
    end
}
```

*[getRandomPivotIndex(i,j) returns a random integer between i and j inclusive in constant time. partition(A,i,j,idx) is the same function used in the simple quick sort algorithm i.e. it rearranges the array (in $O(n)$)so that all the elements smaller than the pivot occur before and all the elements larger than the pivot occur after the pivot. The return value "k" is the final index of the pivot element after partitioning**]***

Answer the following Questions:

a) [8 marks] What is the above code doing? Explain with the help of an array.

It is finding out the maximum element of the array.

b) [2 marks] What is the best case running time of the above algorithm and why?

Best case is $\Theta(n)$, because we have to do partition at-least once.

c) [2 marks] What is the worst case running time of the above algorithm and why?

Worst case is $\Theta(n^2)$, because the partitioning may be "un-balanced" and the recurrence might be: $T(n) = T(n-1) + n$

d) [3 marks] Write down the recurrence equation for the average case running time of the above algorithm (do not solve it).

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

Q-7 [Marks 20] You are given an infinite array A in which the first n cells contain integers in ascending sorted order and the rest of the cells are filled with ∞ (infinity). You are *not* given the value of n (and you can not compute the length of the array using any function). Describe an algorithm that takes an integer x as input and finds a position in the array containing x, if such a position exists, in $O(\lg n)$ time. (If you are disturbed by the fact that the array A has infinite length, assume instead that it is of length n, but that you don't know this length (and you can not compute the length of the array using any function), and that the implementation of the array data type in your programming language returns the error-flag (False) whenever elements A[i] with i > n are accessed.)

Solution:

Basic Idea: Check the element at index k if it is equal to x then return k. Otherwise if A[k] > x then binary search x between A[k/2] and A[k]. However if A[k]<x then check A[2k]. Either you will be able to search x or you will find index k such that A[k] is infinity in which case you will search x between A[k/2] and A[k] using binary search. Now if x is there, the binary search will find it, otherwise returns -1 (x not found). Start the algorithm by taking k = 1.

Analysis: In O(lgn) steps we will reach some index containing infinity in the worst case (what ever the value of n is). After that, the binary search will take O(lgn) steps further. So the total time is lgn + lgn , hence O(lgn).