Practice Quiz 1 Solutions

Problem -1. Recurrences

Solve the following recurrences by giving tight Θ-notation bounds. You do not need to justify your answers, but any justification that you provide will help when assigning partial credit.

(a) $T(n) = T(n/3) + T(n/6) + \Theta(n^{\sqrt{\lg n}})$

Solution: Master method does not apply directly, but we have $T(n) \leq S(n) = 2T(n/3) + \Theta(n^{\sqrt{\lg n}})$. Now apply case 3 of master method to get $T(n) \leq S(n) = \Theta(n^{\sqrt{\lg n}})$. Therefore, we have $T(n) = O(n^{\sqrt{\lg n}})$. Lower bound is obvious.

(b) $T(n) = T(n/2) + T(\sqrt{n}) + n$

Solution: Master method does not apply directly. But $\sqrt{n}$ is much smaller than $n/2$, therefore ignore the lower order term and guess that the answer is $T(n) = \Theta(n)$. Check by substitution.

(c) $T(n) = 3T(n/5) + \lg^2 n$

Solution: By Case 1 of the Master Method, we have $T(n) = \Theta(n^{\log_5(3)})$.

(d) $T(n) = 2T(n/3) + n \lg n$

Solution: By Case 3 of the Master Method, we have $T(n) = \Theta(n \lg n)$.

(e) $T(n) = T(n/5) + \lg^2 n$

Solution: By Case 2 of the Master Method, we have $T(n) = \Theta(\lg^3 n)$.

(f) $T(n) = 8T(n/2) + n^3$

Solution: By Case 2 of the Master Method, we have $T(n) = \Theta(n^3 \log n)$.

(g) $T(n) = 7T(n/2) + n^3$

Solution: By Case 3 of the Master Method, we have $T(n) = \Theta(n^3)$.

(h) $T(n) = T(n-2) + \lg n$

Solution: $T(n) = \Theta(n \log n)$. This is $\sum_{i=1}^{n/2} \lg 2i \geq \sum_{i=1}^{n/2} \lg i \geq (n/4)(\lg n/4) = \Omega(n \lg n)$. For the upper bound, note that $T(n) \leq S(n)$, where $S(n) = S(n-1) + \lg n$, which is clearly $O(n \lg n)$.

Problem -2. True or False

Circle **T** or **F** for each of the following statements, and briefly explain why. The better your argument, the higher your grade, but be brief. No points will be given even for a correct solution if no justification is presented.

(a) T F For all asymptotically positive $f(n)$, $f(n) + o(f(n)) = \Theta(f(n))$.

Solution: True. Clearly, $f(n) + o(f(n))$ is $\Omega(f(n))$. Let $g(n) \in o(f(n))$. For any $c > 0$, $g(n) \leq c(f(n))$ for all $n \geq n_0$ for some n_0. Hence, $g(n) = O(f(n))$, whence $f(n) + o(f(n)) = O(f(n))$. Thus, $f(n) + o(f(n)) = \Theta(f(n))$.

(b) T F The worst-case running time and expected running time are equal to within constant factors for any randomized algorithm.

Solution: False. Randomized quicksort has worst-case running time of $\Theta(n^2)$ and expected running time of $\Theta(n \lg n)$.

(b) T F The collection $\mathcal{H} = \{h_1, h_2, h_3\}$ of hash functions is universal, where the three hash functions map the universe $\{A, B, C, D\}$ of keys into the range $\{0, 1, 2\}$ according to the following table:

x	$h_1(x)$	$h_2(x)$	$h_3(x)$
A	1	0	2
B	0	1	2
C	0	0	0
D	1	1	0

Solution: True. A hash family $\mathcal{H}$ that maps a universe of keys U into m slots is *universal* if for each pair of distinct keys $x, y \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(x) = h(y)$ is exactly $|\mathcal{H}|/m$. In this problem, $|\mathcal{H}| = 3$ and $m = 3$. Therefore, for any pair of the four distinct keys, exactly 1 hash function should make them collide. By consulting the table above, we have:

$h(A) = h(B)$	only for h_3	mapping into slot 2
$h(A) = h(C)$	only for h_2	mapping into slot 0
$h(A) = h(D)$	only for h_1	mapping into slot 1
$h(B) = h(C)$	only for h_1	mapping into slot 0
$h(B) = h(D)$	only for h_2	mapping into slot 1
$h(C) = h(D)$	only for h_3	mapping into slot 0

Problem -3. Short Answers

Give *brief*, but complete, answers to the following questions.

(a) Argue that any comparison based sorting algorithm can be made to be stable, without affecting the running time by more than a constant factor.

Solution: To make a comparison based sorting algorithm stable, we just tag all elements with their original positions in the array. Now, if $A[i] = A[j]$, then we compare i and j, to decide the position of the elements. This increases the running time at a factor of 2 (at most).

(b) Argue that you cannot have a Priority Queue in the comparison model with both the following properties.

- EXTRACT-MIN runs in $\Theta(1)$ time.
- BUILD-HEAP runs in $\Theta(n)$ time.

Solution:

If such priority queues existed, then we could sort by running BUILD-HEAP ($\Theta(n)$) and then extracting the minimum n times ($n.\Theta(1) = \Theta(n)$). This algorithm would sort $\Theta(n)$ time in the comparison model, which violates the $\Theta(n \log n)$ lower bound for comparison based sorting.

(c) Given a heap in an array $A[1 \ldots n]$ with $A[1]$ as the maximum key (the heap is a max heap), give pseudo-code to implement the following routine, while maintaining the max heap property.

DECREASE-KEY(i, δ) – Decrease the value of the key currently at $A[i]$ by δ. Assume $\delta \geq 0$.

Solution:

```
DECREASE-KEY(i, δ)
    A[i] ← A[i] − δ
    MAX-HEAPIFY(A, i)
```

(d) Given a sorted array A of n *distinct* integers, some of which may be negative, give an algorithm to find an index i such that $1 \leq i \leq n$ and $A[i] = i$ provided such an index exists. If there are many such indices, the algorithm can return any one of them.

Solution:

The key observation is that if $A[j] > j$ and $A[i] = i$, then $i < j$. Similarly if $A[j] < j$ and $A[i] = i$, then $i > j$. So if we look at the middle element of the array, then half of the array can be eliminated. The algorithm below (INDEX-SEARCH) is similar to binary search and runs in $\Theta(\log n)$ time. It returns -1 if there is no answer.

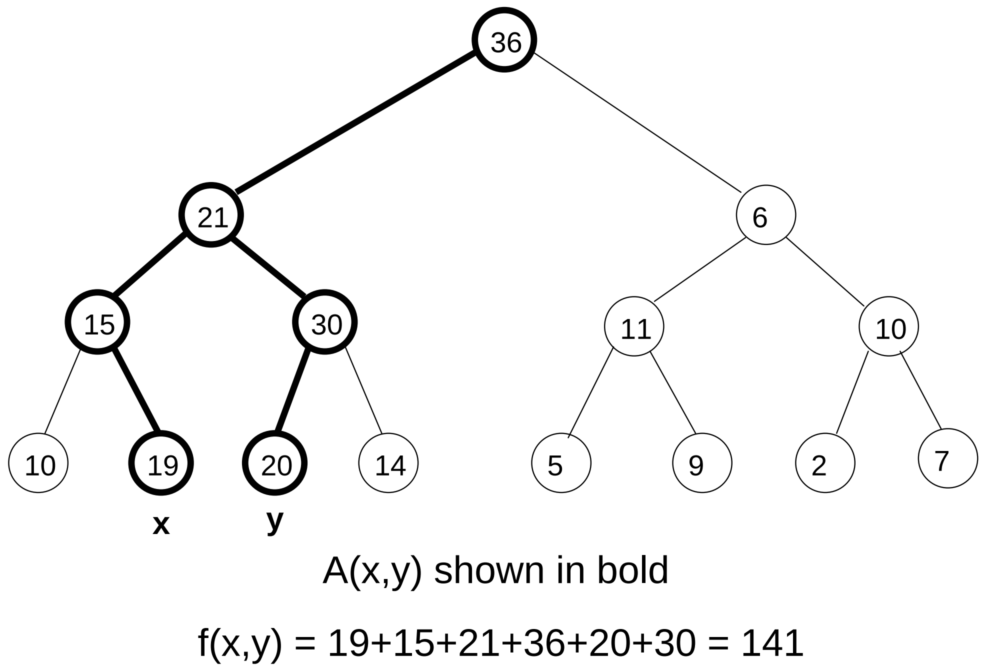
```
INDEX-SEARCH(A, b, e)
    if (e > b)
    return -1
    m = ⌈(e+b)/2⌉
    if A[m] = m
        then return m
    if A[m] > m
        then return INDEX-SEARCH(A, b, m)
        else  return INDEX-SEARCH(A, m, e)
```

Problem -4. Suppose you are given a complete binary tree of height h with $n = 2^h$ leaves, where each node and each leaf of this tree has an associated "value" v (an arbitrary real number).

If x is a leaf, we denote by $A(x)$ the set of ancestors of x (including x as one of its own ancestors). That is, $A(x)$ consists of x, x's parent, grandparent, etc. up to the root of the tree.

Similarly, if x and y are distinct leaves we denote by $A(x, y)$ the ancestors of *either* x or y. That is,

$$A(x, y) = A(x) \cup A(y) \ .$$



Define the function $f(x, y)$ to be the sum of the values of the nodes in $A(x, y)$.

Give an algorithm (pseudo-code not necessary) that efficiently finds two leaves x_0 and y_0 such that $f(x_0, y_0)$ is as large as possible. What is the running time of your algorithm?

Solution:

There are several different styles of solution to this problem. Since we studied divide-and-conquer algorithms in class, we just give a divide-and-conquer solution here. There were also several different quality algorithms, running in $O(n)$, $O(n \lg n)$, and $O(n^2 \lg n)$. These were worth up to 11, 9, and 4 points, respectively. A correct analysis is worth up to 4 points.

First, let us look at an $O(n \lg n)$ solution then show how to make it $O(n)$. For simplicity, the solution given here just finds the maximum value, but it is not any harder to return the leaves giving this value as well.

We define a recursive function $\text{MAX1}(z)$ to return the maximum value of $f(x)$—the sum of the ancestors of a single node—over all leaves x in z's subtree. Similarly, we define $\text{MAX2}(z)$ to be a function returning the maximum value of $f(x, y)$ over all pairs of leaves x, y in z's subtree. Calling MAX2 on the root will return the answer to the problem.

First, let us implement MAX1(z). The maximum path can either be in z's left subtree or z's right subtree, so we end up with a straightforward divide and conquer algorithm given as:

```
MAX1(z)
1  return (value(z) + max {MAX1(left[z]), MAX1(right[z])})
```

For MAX2(z), we note that there are three possible types of solutions: the two leaves are in z's left subtree, the two leaves are in z's right subtree, or one leaf is in each subtree. We have the following pseudocode:

```
MAX2(z)
1  return (value(z) + max {MAX2(left[z]), MAX2(right[z]), MAX1(left[z]) + MAX1(right[z])})
```

Analysis:
For MAX1, we have the following recurrence

$$\begin{aligned} T_1(n) &= 2T_1\left(\frac{n-1}{2}\right) + \Theta(1) \\ &= \Theta(n) \end{aligned} \tag{1}$$

by applying the Master Method.

For MAX2, we have

$$\begin{aligned} T_2(n) &= 2T_2\left(\frac{n-1}{2}\right) + 2T_1\left(\frac{n-1}{2}\right) + \Theta(1) \\ &= 2T_2\left(\frac{n-1}{2}\right) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned} \tag{2}$$

by case 2 of the Master Method.

To get an $O(n)$ solution, we just define a single function, MAXBOTH, that returns a pair—the answer to MAX1 and the answer to MAX2. With this simple change, the recurrence is the same as MAX1

Problem -5. Sorting small multisets

For this problem A is an array of length n objects that has at most k distinct keys in it, where $k < \sqrt{n}$. Our goal is to sort this array in time faster than $\Omega(n \log n)$. We will do so in two phases. In the first phase, we will compute a *sorted* array B that contains the k *distinct* keys occuring in A. In the second phase we will sort the array A using the array B to help us.

Note that k might be very small, like a constant, and your running time should depend on k as well as n. The n objects have satellite data in addition to the keys.

Example: Let $A = \left[5, 10^{10}, \pi, \frac{128}{279}, 10^{10}, \pi, 5, 10^{10}, \pi, \frac{128}{279}\right]$. Then $n = 10$ and $k = 4$.
In the first phase we compute $B = \left[\frac{128}{279}, \pi, 5, 10^{10}\right]$.
The output after the second phase should be $\left[\frac{128}{279}, \frac{128}{279}, \pi, \pi, \pi, 5, 5, 10^{10}, 10^{10}, 10^{10}\right]$.

Your goal is to design and analyse efficient algorithms and analyses for the two phases. Remember, the more efficient your solutions, the better your grade!

(a) Design an algorithm for the first phase, that is computing the sorted array B of length k containing the k distinct keys. The value of k is not provided as input to the algorithm.

Solution:

The algorithm adds (non-duplicate) elements to array B while maintaining B sorted at every intermediate stage. For $i = 1, 2, \ldots, n$, element $A[i]$ is binary searched in array B. If $A[i]$ occurs in B, then it need not be inserted. Otherwise, binary search also provides the location where $A[i]$ should be inserted into array B to maintain B in sorted order. All elements in B to the right of this position are shifted by one place to make place for $A[i]$.

(b) Analyse your algorithm for part (a).

Solution:

Binary search in array B for each element of array A takes $O(\lg k)$ time since size of B is at most k. This takes a total of $O(n \lg k)$ time. Also, a new element is inserted into array B exactly k times, and the total time over all such insertions is $O(1 + 2 + \cdots + k) = O(k^2)$. Thus, the total time for the algorithm is $O(n \lg k + k^2) = O(n \lg k)$ since $k < \sqrt{n}$.

(c) Design an algorithm for the second phase, that is, sorting the given array A, using the array B that you created in part (a). Note that since the objects have satellite data, it is not sufficient to count the number of elements with a given key and duplicate them.
Hint: Adapt Counting Sort.

Solution:

Build the array C as in counting sort, with $C[i]$ containing the number of elements in A that have values less than or equal to $B[i]$. Counting sort will not work as is since

$A[i]$ is necessarily an integer. Or, it may be some integer of very large value (there is no restriction on our input range). Therefore $A[i]$ is an invalid index into our array C. What we would like to do is assign an integral "label" for the value $A[i]$. The label we choose is the index of the value $A[i]$ in the array B calculated in the last part of the problem.

How do we find this index? We could search through B from beginning to end, looking for the value $A[i]$, then returning the index of B that contains $A[i]$. This would take $O(k)$ time. But, since B is already sorted, we can use BINARY-SEARCH to speed this up to $O(\log k)$. Let BINARY-SEARCH(S, x) be a procedure that takes a sorted array S and an item x within the array, and returns i such that $S[i] = x$. The modified version of COUNTING SORT is included below, with modified lines in bold:

COUNTING-SORT(A)
/ Uses Arrays* $C[1..k]$, $D[1..k]$, *and* A*-out*$[1..n]$ */*
For $i = 1$ to k do $C[i] \leftarrow 0$; */* Initialize */*
For $i = 1$ to n do */* Count number of elements */*
    Location $\leftarrow$ BINARY-SEARCH$(B, A[i])$;
    C[Location] $\leftarrow$ C[Location] $+1$;
$D[1] \leftarrow C[1]$;
For $j = 2$ to k do */* Build cumulative counts */*
    $D[j] \leftarrow D[j-1] + C[j]$;
For $i = n$ downto 1 do */* Construct Sorted List A-Out */*
    Location $\leftarrow$ BINARY-SEARCH$(B, A[i])$;
    Out-Location $\leftarrow$ D[Location];
    D[Location] $\leftarrow$ D[Location] -1;
    A-out[Out-Location] $\leftarrow A[i]$;
Output(A-out);

(d) Analyse your algorithm for part (c).

Solution:

The running time of the modification to COUNTING-SORT we described can be broken down as follows:

- First Loop: $O(k)$.
- Second Loop: $O(n)$ iterations, each iteration performing a BINARY-SEARCH on an array of size k. Total Work: $O(n \log k)$.
- Third Loop: $O(k)$.
- Fourth Loop: $O(n)$ iterations, each iteration performing a BINARY-SEARCH on an array of size k. Total Work: $O(n \log k)$.

The running time is dominated by the second and fourth loops, so the total running time is $O(n \log k)$.