Quiz 1 Solutions

Problem 1. Asymptotic orders of growth [9 points] (3 parts)

For each of the three pairs of functions given below, rank the functions by increasing order of growth; that is, find any arrangement g_1, g_2, g_3 of the functions satisfying $g_1 = O(g_2)$, $g_2 = O(g_3)$. $\lg n$ represents logarithm of n to the base 2.

(a)

$$f_1(n) = 8\sqrt{n}, \quad f_2(n) = 25^{1000}, \quad f_3(n) = (\sqrt{3})^{\lg n}$$

Solution: $f_2(n), f_1(n), f_3(n)$

(b)

$$f_1(n) = \tfrac{1}{100}, \quad f_2(n) = \tfrac{1}{n}, \quad f_3(n) = \tfrac{\lg n}{n}$$

Solution: $f_2(n), f_3(n), f_1(n)$

(c)

$$f_1(n) = 2^{\lg^3 n}, \quad f_2(n) = n^{\lg n}, \quad f_3(n) = \lg n!$$

Solution: $f_3(n), f_2(n), f_1(n)$

Problem 2. Balanced Augmented BST [20 points] (2 parts)

In this question, we use balanced binary search trees for keeping a directory of MIT students. We assume that the names of students have bounded constant length so that we can compare two different names in $O(1)$ time. Let n denotes the number of students.

(a) Say we have a binary search tree with students' last names as the keys with lexicographic dictionary ordering. Let k be the number of students whose last name starts with a given prefix x.

For example, if the tree contains 5 names {ABC, ABD, ADA, ADB, ADC} there are k=2 names, ABC and ABD respectively, starting with the prefix x=AB. How can we output a list of all those students in $O(k + \log n)$ time?

Solution: Consider the following procedure for traversing the tree.

```
traverse(tree, prefix):
    if tree is empty then
        return
    name := the last name at the root of tree
    if prefix is a prefix of name then
        traverse(left subtree of tree, prefix)
        output(name)
        traverse(right subtree of tree, prefix)
    elseif prefix < name then
        traverse(left subtree of tree, prefix)
    else
        traverse(right subtree of tree, prefix)
```

It suffices to run `traverse`(the entire balanced BST, x) to find all last names with prefix x. The algorithm has the following properties:

- It finds all names with prefix x, because it never excludes a subtree that could contain a name with prefix x.
- It only outputs names with x as a prefix.
- The running time is $O(k+\log n)$, because at each depth it visits at most two nodes for which x is not a prefix. Therefore, it visits at most $2 \cdot O(\log n) + k$ nodes.

Another correct solution: Another correct solution is to find the lexicographically first last name with prefix x, and then to continue visiting consecutive nodes in lexicographic order as in the in-order tree traversal, until we encounter a name for which x is not a prefix. This procedure visits exactly the same nodes as the first solution, and also runs in $O(k + \log n)$ time.

Common errors:

- It is not true that the nodes with prefix x must constitute a connected component in the tree. As a counterexample consider the following tree:

```
      ABZ
     /   \
   AAA   ZZZ
     \
     ABA
```

  The nodes with prefix `AB` are not connected.
- Finding a successor of a node in the tree does not always take $O(1)$ time. It can take as much as $O(\log n)$ time, but for a balanced tree finding the consecutive k keys can be shown to take only $O(\log n + k)$ time.

(b) Give an algorithm to return the number of nodes k with names starting with a given prefix x in $O(\log n)$ time. You are allowed to augment the binary search tree nodes.

Solution: We can use a similar augmentation of Number of nodes in the left subtree and right subtree which we used for finding the Rank of a node (in PS 2 and lecture/recitation). First traverse down the tree to find a node x_{first} which starts with the prefix x. Then traverse the left subtree of x_{first} to find the minimum node x_{min}(lexicographically) which starts with prefix x. Traverse the right subtree of x_{first} to find the largest node x_{max} (lexicographically) that starts with prefix x. x_{first}, x_{min} and x_{max} can be found in $O(\log n)$ time. Using the augmentation we can find the ranks of x_{min} and x_{max}. The required number of nodes is equal to $\mathsf{Rank}(x_{max}) - \mathsf{Rank}(x_{min}) + 1$.

Another possible augmentation is for each node keeping track of number of nodes in its subtree which share a common prefix with it for all possible prefixes of that node. e.g A node with key ABCDE will keep track of number of nodes in its subtree that start with prefixes A, AB, ABC, ABCD and ABCDE. Since the words are of constant length, there are constant number of prefixes possible for each node. Now to find the number of nodes which start with prefix x, we traverse the tree to find the first node x_{first} which starts with prefix x. The required number of nodes is the value stored in the augmentation of x_{first} for the prefix x. In both cases, it is easy to see that only nodes on the path traversed while insertion are required to be updated for augmentation value which takes $O(\log n)$ time.

Problem 3. Employee Dictionary [15 points] (1 part)

We are interested in building a phonebook for the employees of a small company of size < 64, using a *dictionary*. Our hash table will have size $m = 64$, and the employees' last names will be used as keys.

The longest employee name is 15 characters long, so to simplify our task we convert every last name into a 15-character string by introducing *space characters* in front of the first character. For instance, the last name 'Devadas' will be padded with 8 space characters preceding the leading 'D'. Now we encode every letter of the English alphabet—together with the space character—into a unique 6-bit binary number, so that every last name can be represented uniquely by a 90-bit binary number.

Consider the following hash functions for our dictionary. Which of them is likely to perform the best? Justify your choice: For the hash functions you rejected, describe a collection of last names for which the hash function performs poorlyand explain why. For the hash function of your choice, explain why it is better than the other hash functions. The input k to the following functions is the 90-bit binary number corresponding to a last name:

1. $h(k) = k \mod 64$;
2. $h(k) = (a \cdot k \bmod 2^{18}) >> 12$, for $a = 4097$;
3. $h(k) = (a \cdot k \bmod 2^{90}) >> 84$, for $a = 1 + 2^6 \cdot 54605$;

Note: Observe that $4097 = 1 + 2^{12}$ and $54605 = 1101010101001101_2$. The $>>$ operator is the right shift operatr.

Solution:

1. Note that $64 = 2^6$. Hence, the hash function keeps the 6 least significant bits of k, which correspond to the last character of the name—since every character is encoded using 6 bits. E.g., if the last name is 'Devadas', then the value of the hash function will be the 6-bit binary number (in decimal, a number from 0 through 63) encoding the character 's'. So, names ending in the same letter are hashed to the same number using this hash function. E.g. 'Clinton', 'Franklin', 'Jefferson', etc. are all hashed to the same number.
2. Since $a = 1 + 2^{12}$, the operation $a \cdot k$ performs the addition of k with $k' := 2^{12}k$, which is just k shifted left (as a binary number) by 12 positions. Then the operation $\bmod 2^{18}$ followed by the operation $>> 12$ keeps the bits 13 through 18 of $k + k'$ (counting from the least significant bit, which is taken to be the 1st bit). See Figure 1. If e_1 is the number encoding the last character of the name and e_3 the number encoding the third-to-last character, it can be shown (see Figure 1) that $h(k) = (e_1 + e_3) \bmod 64$—it is important to note here that there is no carry from position 12 to position 13 when performing the addition of k and k'. Hence, last names sharing the same last and third-to-last letters are hashed to the same number. E.g. 'Jefferson' and 'Petersen' are hashed to the same number.

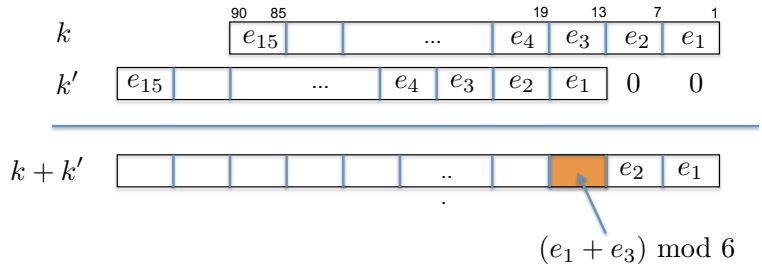

Figure 1: The second hash function.

3. Given the value of a selected for this function, the operation $a \cdot k$ performs the addition $S := k_0 + k_6 + k_8 + k_9 + k_{12} + k_{14} + k_{16} + k_{18} + k_{20} + k_{21}$, where k_i is the input key k shifted left (as a binary number) by i bits. Then the operation $\bmod 2^{90}$ followed by the operation $>> 84$ selects the bits 85 through 90 of the result. The choice of a here looks better from the previous choice, because (i) there are many shifted copies of k created, which are going to be added to produce the resulting hash, and (ii) the shifts are not in multiples of 6 bits which is the length used to represent a single character (so there is also mixing at the granularity of the encoding of characters). Moreover, the function keeps the most significant positions of S, which allows for the outcomes of the mixing to be visible to the hash value.

   However, this function may also suffer from poor performance, because the shifts are relatively small compared to the length of the key (the maximum shift is by 21 binary positions, whereas the length of the key is 90 bits). In order for this function to perform better than the previous function, it should be the case that either most of the last names are long (longer than say 12 characters), or that the binary encoding of the letters is selected very carefully. Consider, e.g., what would happen if most of the names had length ≤ 9 characters and the binary encoding of the space character was 000000_2. It is easy to see that $S \leq 10 \cdot k_{21}$, since k_{21} is the largest summand in the summation S. So, if the space character was encoded by 0 and a last name had ≤ 9 characters, then the most significant non-zero bit of k would be at position ≤ 54. This would imply that the most significant non-zero bit of k_{21} would be at position ≤ 75 (since k_{21} is k shifted left by 21 positions), which would imply that the most significant non-zero bit of S is at position ≤ 79 (since $S \leq 10 \cdot k_{21}$ and $10 \leq 2^4$). So the bits 85 through 90 of S would all be zero. So all names of length ≤ 9 characters would be hashed to position 0.

Problem 4. Asymptotic Runtime Analysis [15 points] (1 part)

Professor Devadas has an idea to assign students to recitations based on their Quiz 1 grades. He wants to split all N students into 4 recitations such that the maximum difference between the Quiz 1 grades of any two students in the same recitation is as small as possible. To figure out if this idea is reasonable, he wants to know how small the staff can make this maximum difference.

Let the number of students be N and suppose that Quiz 1 scores are all integers in the range $1 \ldots M$. Professor Daskalakis suggests the following Python code, which takes the value of M and a pre-sorted list of quiz scores:

```
def assign_recitations(M, scores):
    N = len(scores)
    a = 1
    b = M
    while a < b:
        c = (a+b)/2
        groups = 1
        group_low = scores[0]
        for i in range(N):
            if scores[i] > group_low + c:
                groups = groups + 1
                group_low = scores[i]
        if groups <= 4:
            b = c
        else:
            a = c+1
    return a
```

What is the asymptotic running time of this algorithm? Express your answer as a function of N and/or M using Θ-notation.

Solution: The inner loop (the `for` loop) runs in constant time per iteration with N iterations. The outer loop is performing binary search on the interval $[1, M]$ and thus takes $\log_2 M$ iterations. Thus, the runtime is $\Theta(\log M) \cdot \Theta(N) \cdot \Theta(1) = \Theta(N \log M)$.

Problem 5. Airplane Scheduling [20 points] (1 part)

Logan airport management asks you to compute the airport "load" during one day, which is the maximum number of planes that are on the ground at the same time. You are provided the data for n airplanes' arrival and departure times. To simplify computation, the day is divided into m time segments and only these segments are recorded. For the plane i, $start[i]$ denotes the time segment of arrival and $end[i]$ denotes the time segment of departure; $1 \leq start[i] \leq end[i] \leq m$. A plane is considered to be on the ground during arrival and departure time segments, as well as all segments in between. For example, if $n = 5$, $m = 10$ and planes' $(arrival, departure)$ pairs are $\{(5,7),(1,3),(8,10),(2,5),(4,9)\}$, then the airport load is 3 (during time segment 5 planes 1, 4, and 5 are on the ground).

Give an $O(m+n)$ time algorithm to find the airport load. If you provide an algorithm with worse time complexity you will be given partial credit.

Solution:

```
/* Lists to keep track of arrivals and departures */
for(i = 1; i <= m; i++)
  A[i]= D[i] = 0

/* Go through (arrival, departure) pairs, incrementing count */
for(j = 1; j <= n; j++) {
  A[arrival[j]] = A[arrival[j]] + 1
  D[departure[j]] = D[departure[j]] + 1
}

/* Compute current load and compare with max load */
max_load = 0; load = 0;
for(i = 1; i < m; i++) {
  load = load + A[i];
  if (load > max_load) max_load = load;
  load = load - D[i];
}
```

The first loop is $O(m)$. The second loop is $O(n)$. The third loop is $O(m)$. Note that the `max_load` check has to happen after incrementing load by the number of arrivals and before decrementing by the number of departures, in order to take into acount the fact that a plane is considered to be on the ground during its departure segment.

The above two arrays can be merged into a single array `C`. We will increment `C[arrival[j]]` and decrement `C[departure[j]+1]` in the second loop. Then, we keep adding the entries of `C` and find the maximum sum, which will be the maximum load. The `+1` added to `departure[j]` ensures that we treat a plane as being on the ground during its departure segment.

Problem 6. Multiplying two N-bit Numbers [20 points] (2 parts)

Let X and Y be two n-bit numbers in base B and we would like to compute their product $Z = XY$.
A naive divide-and-conquer technique can work as follows:
Divide X into two $(n/2)$-bit numbers X_1 and X_0 and similarily Y into Y_1 and Y_0. Let $b = B^{n/2}$.

$$X = X_1 b + X_0$$
$$Y = Y_1 b + Y_0$$
$$Z = (X_1 b + X_0)(Y_1 b + Y_0)$$
$$Z = Z_2 b^2 + Z_1 b + Z_0$$

where $Z_2 = X_1 Y_1, Z_1 = X_1 Y_0 + X_0 Y_1$ and $Z_0 = X_0 Y_0$ can be obtained recursively. Assume addition of two n-bit numbers takes $\Theta(n)$ time. Also assume that multiplying an n-bit number by B^m also takes $\Theta(n + m)$ time.

(a) Prove that the above naive divide-and-conquer algorithm runs in $\Theta(n^2)$ time.

Solution: Let $T(n)$ denote the running time of the algorithm on n-bit numbers. It consists of several computations. Splitting X into X_1 and X_0, as well as splitting Y into Y_1 and Y_0, takes $\Theta(n)$ time. Computing X_1Y_1, X_1Y_0, X_0Y_1 and X_0Y_0 recursively takes $4T(\frac{n}{2})$ time, as each subproblem performs multiplication of two $\frac{n}{2}$-bit numbers. Adding X_1Y_0 and X_0Y_1 to compute Z_1 takes $\Theta(n)$ time, as it is addition of two n-bit numbers. In the last step, computing $Z = Z_2b^2 + Z_1b + Z_0$ takes $\Theta(n)$ time because Z_2b^2 is multiplication of n-bit number with $b^2 = B^n$, Z_1b is multiplication of n-bit number with $b = B^{\frac{n}{2}}$, and there is addition of three numbers of size at most $2n$. Finally, we can write:

$$T(n) = 4T\left(\frac{n}{2}\right) + f(n),$$

where $f(n) = \Theta(n)$. We can apply Master Theorem to solve this recurrence.

$$a = 4, b = 2, \log_b a = \log_2 4 = 2, n^{\log_b a} = n^2.$$

Therefore, $f(n) = \Theta(n) = O(n^{2-\epsilon})$ for any $0 < \epsilon \leq 1$ (take $\epsilon = 1$ for example). From case 1 of Master Theorem it follows that $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$.

(b) It turns out that we only need 3 multiplications to compute Z_2,Z_1 and Z_0 instead of 4, required in the previous case, at the cost of some extra additions as follows:
$Z_2 = X_1Y_1$, $Z_0 = X_0Y_0$ and $Z_1 = (X_1 + X_0)(Y_1 + Y_0) - Z_2 - Z_0$
What is the time complexity of this algorithm in terms of n?

Solution: There are three subproblems in the modified algorithm. Computing X_1Y_1, X_0Y_0 and $(X_1+X_0)(Y_1+Y_0)$ recursively takes $3T(\frac{n}{2})$ time, while all other operations take $\Theta(n)$ time (there are two subtractions in computing Z_1, while Z is computed as before). Now, the recurrence is:

$$T(n) = 3T\left(\frac{n}{2}\right) + f(n),$$

where $f(n) = \Theta(n)$. We can again apply Master Theorem.

$$a = 3, b = 2, \log_b a = \log_2 3, n^{\log_b a} = n^{\log_2 3}.$$

Therefore, $f(n) = \Theta(n) = O(n^{\log_2 3 - \epsilon})$ for any $0 < \epsilon \leq \log_2 3 - 1$ (there exists such ϵ because $\log_2 3 > 1$). Again, from case 1 of Master Theorem it follows that $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 3})$.
Note that this algorithm is better then the previous one, since $\log_2 3 < 2$ and, correspondingly, $\Theta(n^{\log_2 3})$ is lower asymptotic time than $\Theta(n^2)$.