The Australian National University Semester 2, 2024
School of Computing **Midterm Assessment**

Convenor(s): Ahad N. Zehmakan & Yun Kuen Cheung

Algorithms (COMP3600/6466)

Please Write Your Information.

- Family Name:
- University ID:

PLEASE READ BEFORE WORKING ON THE SOLUTION:

- You are ONLY allowed to use the following items during the exam: (1) your exam sheets, (2) pen.
- You have **110 minutes** to solve the exercises.
- This exam consists of 4 exercises and the total score is 100 points. Please make sure to use your time wisely. You are strongly encouraged to first read all the questions.
- Pencils are not allowed. Pencil-written solutions will not be marked.
- Provide only one solution to each exercise. Please clearly mark the solution that should be graded.
- Pages 9 to 13 are blank. You can use them to write your solutions if you need more space, but please clearly mark which exercise the solution belongs to. You are allowed to write on both front and back of each sheet.
- You can ask for scrap papers. However, note that the scrap papers are NOT marked. Please write your university ID on all your scrap papers.
- This assessment will contribute 40% to your overall grade for the course.
- You may use anything that has been introduced and proved in the lectures and weekly exercises. You do not need to re-prove it, but you need to state it clearly and concretely. However, if you need something different than what we have shown, you must write a new proof or at least list all necessary changes.
- All logs should be treated as base 2.

Exercise 1 **Multiple Choice** **36 points**

Please answer the following questions. You will **gain 4.5 points** for each correctly answered question. You do NOT lose points for wrong answers.

Use an X (cross) to select your choice. Note that this is a **single-choice** exercise, i.e., only one of all four choices can and should be selected. Selecting more than one will be marked as wrong, even if the correct answer is among your choices. If you change your mind, cross out your selected answer completely, and circle the other (that you want to select). If your selection is unclear to us, you will receive 0 points.

1. Let $T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + 4n$ for $n \geq 2$ and $T(1) = 7$. Which statement is correct?

   (a) $T(n) = \Theta(n)$
   (b) $T(n) = \Theta(n \log(n))$
   (c) $T(n) = \Theta(n^2)$
   (d) $T(n) = \Theta(n^2 \log(n))$

   Solution

   (b) We use the Master Theorem. Here $\log_b a = \log_2 2 = 1$. Since $4n = \Theta(n^{\log_a b} = n)$, by case 2 of the Master Theorem, we have $T(n) = \Theta(n \log(n))$.

2. Let $T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + n$ for $n \geq 4$ and $T(1) = T(2) = T(3) = 1$. Which statement is correct?

   (a) $T(n) = \Theta((\log_3(n))^4)$
   (b) $T(n) = \Theta(n)$
   (c) $T(n) = \Theta(n \log(n))$
   (d) $T(n) = \Theta(n(\log(n))^2)$

   Solution

   (b) We use the Master Theorem. Here $\log_b a = \log_4 3 \simeq 0.79$. Since $4n = \Omega(n^{(\log_a b)+\epsilon})$ for $\epsilon = 0.1$, and $3 \times \frac{n}{4} \leq cn$ for some $\frac{3}{4} \leq c < 1$ by case 3 of the Master Theorem, we have $T(n) = \Theta(n)$.

3. Recall the Insertion Sort algorithm. Suppose the input is an array A of integers $1, 2, \cdots, n$. The input is selected at random in the following way: with probability $\frac{1}{2}$ the input is sorted in ascending order (our desired order) and with probability $\frac{1}{2}$, it is chosen uniformly and independently at random from all remaining $n! - 1$ permutations. For example, for $n = 3$, the input is $[1, 2, 3]$ with probability $\frac{1}{2}$ and the probability of each of the $3! - 1 = 5$ remaining permutations is $\frac{1}{2} \times \frac{1}{5} = \frac{1}{10}$. What is the expected run time of Insertion Sort in this setting?

   (a) $\Theta(n)$
   (b) $\Theta(n \log(n))$
   (c) $\Theta(n^2)$
   (d) $\Theta(1)$

   Solution

   (c) If the input is sorted in the desired order, then the Insertion Sort algorithm takes $\Theta(n)$ time. This is because every time an element is picked it's compared with the element right before it and we observe that no swap is needed. Thus, it's put back in its original position.

   If the input is chosen uniformly at random among the remaining $n! - 1$ permutations, very similar to the discussion in the lecture, we can show that the expected run time is $\Theta(n^2)$.

   Since with probability 1/2 the time required is $\Theta(n^2)$, the overall expected run time is also in

$\Theta(n^2)$.

4. Recall the coupon collector problem from the lecture, where we wanted to collect at least one coupon from each of the n types. Consider the following two scenarios:

   Scenario 1: We just want to collect $\frac{n}{2}$ coupons of different types (instead of n in the original setup).
   Scenario 2: We want to collect at least two coupons from each type (instead of one in the original setup).

   What is the expected number of coupons bought until we reach our goal in each of these scenarios?

   (a) $\Theta(n)$ in scenario 1 and $\Theta(n^2)$ in scenario 2.

   (b) $\Theta(n \log(n))$ in scenario 1 and $\Theta(n \log(n))$ in scenario 2.

   (c) $\Theta(n)$ in scenario 1 and $\Theta(n \log(n))$ in scenario 2.

   (d) $\Theta(n \log(n))$ in scenario 1 and $\Theta(n^2)$ in scenario 2.

   Solution

   (c) For Scenario 1, with a similar argument as in the lecture we have that the expected number of coupons needs to be collected is equal to

   $$\sum_{i=1}^{n/2} \frac{n}{n-i+1} \leq \sum_{i=1}^{n/2} \frac{n}{(n/2)+1} \leq \sum_{i=1}^{n/2} 2 = O(n)$$

   Note that the number of collected coupons is trivially in $\Omega(n)$ since we need to collect at least $n/2$ coupons. Thus, the expected number is in $\Theta(n)$.

   For Scenario 2, the expected number is trivially in $\Omega(n \log n)$ since even collecting one coupon for each type requires $\Omega(n \log n)$ coupons as discussed in the lecture. Furthermore, the number of required coupons is in expectation in $O(n \log n)$ because we can in the first phase wait until we have one from each type and then in the second phase wait again to have another coupon for each type. This is clearly an upper bound, since we ignore the extra coupons from the first phase. From the lecture, each of these two phases takes at most $O(n \log n)$. Thus, overall $O(n \log n)$ trials suffice.

5. Recall the egg dropping problem where you want to find the highest floor that if you drop the egg, it does not break. Suppose you have an unlimited number of identical eggs and the building has 64 floors. What is the number of drops executed by the binary search based algorithm (explained in the lecture) in the worst case scenario?

   (a) 6

   (b) 7

   (c) 11

   (d) 64

   Solution

   (b) One can check that in the worst case, the binary search based algorithm, discussed in the lecture, needs $(\log_2 n) + 1$ trials and this happens when the solution is $n-1$ or n. To see that the answer is $(\log_2 n) + 1$ and not $\log_2 n$, it might be easier to check a smaller example, such as $n = 4$.

6. Consider an array A with k_1 number of 1's, k_2 number of 2's, and k_3 number of 3's such that $k_1+k_2+k_3 = n$, where n is the size of the array A. For example, for the array $A = [2, 3, 3, 2, 1, 2, 1]$, we have $k_1 = 2$, $k_2 = 3$, and $k_3 = 2$.

   Consider the following simple randomized algorithm, which aims to find an index of A where there is a 1. (Function rand$(1, n)$ returns an integer from 1 to n uniformly and independently at random.)

Algorithm 1 FindOne Algorithm (A)

```
x ← rand(1, n)
while A[x] ≠ 1 do
    x ← rand(1, n)
end while
return x
```

   What is the expected run time of FindOne?

   (a) $\Theta(k_1)$

   (b) $\Theta\left(\frac{n}{k_1}\right)$

   (c) $\Theta\left(\frac{k_1}{n}\right)$

   (d) $\Theta\left(\frac{k_2+k_3}{n}\right)$

> **Solution**
>
> **(b)** The process of randomly selecting indices and checking their values follows a geometric distribution with a success probability of $\frac{k_1}{n}$. For a geometric distribution with success probability p, the expected number of trials needed to achieve the first success is given by $\frac{1}{p}$. In this case, this translates to $\frac{n}{k_1}$.

7. Which statement is correct about the stability property of the sorting algorithms?

   (a) RandQuick Sort is a stable algorithm.

   (b) Counting Sort is NOT a stable algorithm.

   (c) Any Comparison based sorting algorithm can be made stable with the same modification scheme.

   (d) None of the above statements are correct.

> **Solution**
>
> **(c)** We achieve this by converting each element $A[i]$ into the pair $(A[i], i)$, comparing the first element of the pair, and in the case of a tie, comparing the second element, as discussed in the weekly exercises. (a) and (b) are wrong as disscused in the lecture.

8. Which statement is true about the RadixSort algorithm given below? (For a number such as 476, 6 is digit 1, 7 is digit 2, and 4 is digit 3.)

Algorithm 2 RadixSort (A)

```
d ← maximum number of digits in any number in array A
for i = 1 to d do
    Sort array A on digit i using Counting Sort
end for
```

   (a) The algorithm is still correct if we iterate from $i = d$ to $i = 1$ in the **for** loop instead of $i = 1$ to $i = d$.

(b) The algorithm is still correct if we iterate from $i = 2$ to $i = d$ instead of $i = 1$ to $i = d$.

(c) The algorithm is still correct if we use RandQuick Sort instead of Counting Sort.

(d) Radix Sort is stable.

Solution

(d) is correct as discussed in the lecture. (c) is wrong since RandQuick sort is not stable. For (a) and (b) it is easy to build examples where the algorithm wouldn't work.

Exercise 2 **Asymptotic Analysis** **24 points**

In this exercise, we suppose that all functions correspond to run times of algorithms and thus are non-negative. Each part is worth **6** points.

1. Let $f(n) = (\log(n))^3$ and $g(n) = \log(n^7)$. What is the correct relation between $f(n)$ and $g(n)$: $f(n) = o(g(n))$, $f(n) = \omega(g(n))$ or $f(n) = \Theta(g(n))$? Prove your answer.

2. Let $f(n) = 2^{\log(\log(n^2))}$ and $g(n) = \sqrt{n}$. What is the correct relation between $f(n)$ and $g(n)$: $f(n) = o(g(n))$, $f(n) = \omega(g(n))$ or $f(n) = \Theta(g(n))$? Prove your answer.

3. Let $f(n) = \sum_{i=1}^{n} \frac{(\log(n))^3}{i}$ and $g(n) = 4^{\log((\log(n))^2)}$. What is the correct relation between $f(n)$ and $g(n)$: $f(n) = o(g(n))$, $f(n) = \omega(g(n))$ or $f(n) = \Theta(g(n))$? Prove your answer.

4. Is the statement below correct? If yes, provide a proof. If no, give a counter-example.

   For any three non-negative functions $f(n)$, $g(n)$ and $h(n)$: if $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.

Solution

1. We have

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{(\log(n))^3}{\log(n^7)} = \lim_{n\to\infty} \frac{(\log(n))^3}{7\log(n)} = \lim_{n\to\infty} \frac{(\log(n))^2}{7} = \infty,$$

which yields that $f(n) = \omega(g(n))$.

2. Note that $f(n) = 2^{\log(\log(n^2))} = \log(n^2) = 2\log n$. Similar calculation

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{2\log(n)}{\sqrt{n}} = \frac{\infty}{\infty},$$

by applying the L'Hôpital's rule we will have

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{4n^{1/2}}{n\ln(2)} = \lim_{n\to\infty} \frac{4}{n^{1/2}\ln(2)} = 0,$$

which yields that $f(n) = o(g(n))$.

3. Note that $g(n) = 4^{\log((\log(n))^2)} = ((\log(n))^2)^{\log_2(4)} = ((\log(n))^2)^2 = (\log(n))^4$. Also, note that

$$f(n) = \sum_{i=1}^{n} \frac{(\log(n))^3}{i} = (\log(n))^3 \sum_{i=1}^{n} \frac{1}{i} = (\log(n))^3(\log(n) + c).$$

We used that the sum is harmonic numbers as discussed in the lecture.

Hence,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{(\log(n))^3(\log(n) + c)}{(\log(n))^4} = 1,$$

which yields that $f(n) = \Theta(g(n))$.

4. For this part, we will follow the definition of O.

As $f(n) = O(g(n))$, then there exist positive constants c_0 and n_0 such that for any $n \geq n_0$, we have $0 \leq f(n) \leq c_0 g(n)$.

As $g(n) = O(h(n))$, then there exist positive constants c_1 and n_1 such that for any $n \geq n_1$, we have $0 \leq g(n) \leq c_1 h(n)$.

Combining the above two statements, we can conclude that for any $n \geq \max(n_0, n_1)$, we have $f(n) \leq c_0 c_1 h(n)$. This is equivalent to the definition of $f(n) = O(h(n))$ for the choice of constants $c' = c_0 c_1$ and $n' = \max(n_0, n_1)$.

Exercise 3 **Recurrence Analysis** **15 points**

Use the **recursion tree method** to determine the growth rate of the following recurrence formula: $T(n) = T\left(\lfloor \frac{n}{5} \rfloor\right) + T\left(\lceil \frac{4n}{5} \rceil\right) + 4n$ for $n \geq 5$ and $T(n) = 2$ for $1 \leq n \leq 4$.

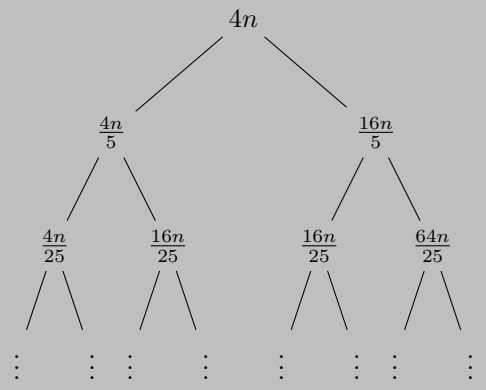
Note 1. Provide your answer in Θ notation, e.g. $\Theta(n \log n)$.

Note 2. There is NO need to provide a proof by induction.

Note 3. You MUST use the recursion tree method.

Solution

The recursion tree for the given problem looks as follows (note here we will ignore the floor and ceil, rounding up/down. This is OK since we are just making a guess when using recursion tree and the proof is done via induction, which you were not asked to do in the exam):



From the above tree it seems that the total cost at each level adds up to $4n$.
Now, we need to calculate the total number of levels in order to generate an expression for the total cost. Since our tree is not balanced, we have for the leftmost branch (the branch where problem size is being divided by 5 at each subsequent level):

$$n(\frac{1}{5})^k = 1 \Rightarrow k = \log_5 n \tag{1}$$

For our tree, this leftmost branch is the branch with the least depth out of all the other branches. Hence the total cost up-to (and including) the level corresponding to $k = \log_5 n$ will give us a lower bound on the total cost for the recursion. Hence we get,

$$\sum_{k=0}^{\log_5 n} 4n = 4n(\log_5 n + 1) \Rightarrow T(n) = \Omega(n \log n) \tag{2}$$

Now, similarly for the rightmost branch we have,

$$n(\frac{4}{5})^k = 1 \Rightarrow k = \log_{\frac{5}{4}} n \tag{3}$$

Note at this level all of the other branches would have already terminated, hence the expression $\sum_{k=0}^{\log_{\frac{5}{4}} n} 4n$ will provide us with an upper bound for the total cost of the recursion. Therefore we have,

$$\sum_{k=0}^{\log_{\frac{5}{4}} n} 4n = 4n(\log_{\frac{5}{4}} n + 1) \Rightarrow T(n) = O(n \log n) \tag{4}$$

Now, from (2) and (4) we have:

$$T(n) = O(n \log n) \ \ \& \ \ T(n) = \Omega(n \log n) \Rightarrow T(n) = \Theta(n \log n) \tag{5}$$

Exercise 4 **Algorithm Design** **25 points**

Consider the following problem:

Input: An array A of size n and an array B of size $\frac{n}{\sqrt{\log(n)}}$. All elements in both A and B are positive integers with at most $\sqrt{\log(n)}$ digits. Array A is sorted while B is NOT sorted.

Output: Return "YES" if every element in B is also in A. Return "NO" otherwise.

- Design an algorithm to solve this problem.
- Analyze the run time of your proposed algorithm in Θ notation and justify your answer.
- Justify why your algorithm always returns a correct solution.

Note 1. If an element appears k times in B, the problem requires that it appears at least k times in A as well. For example, the answer for $A = [1, 1, 2, 3]$ and $B = [2, 1, 1]$ is YES, but the answer for $A = [1, 2, 3]$ and $B = [2, 1, 1]$ is NO.

Note 2. To receive the full mark for this exercise, you need to provide an algorithm whose run time is in $O(n)$. However, you may receive partial points for an algorithm which runs in $\Theta(n\sqrt{\log(n)})$ (up to 60%) and in $\Theta(\frac{n^2}{\sqrt{\log(n)}})$ (up to 30%).

Note 3. You do not necessarily need to provide pseudocode for your algorithm, but it should be completely clear what the algorithm does step by step from your explanations.

Algorithm 3 Solution (A,B)

```
Sort array B using Radix Sort with Counting Sort
i ← 1
j ← 1
while i ≤ length(B) do
    if j > length(A) then
        Return "NO"
    else if B[i] > A[j] then                ▷ Ignore excess elements in A
        j++
    else if B[i] == A[j] then               ▷ The element in B has been found in A
        i++
        j++
    else                                    ▷ The element in B is not in A
        Return "NO"
    end if
end while
Return "YES"
```

Solution

Motivation:

Refer to Algorithm 3 for the solution. Essentially, the solution first sorts B and then for each element in B, attempts to match with an element in A.

To understand how we arrive here, we first note that if we were asked to solve the problem by hand, the natural thing to do would be to group up all the same elements of the array B and then verify if each group appears in A. The easiest way to do this is to simply sort the array.

To understand the second part of the algorithm, we consider an example consisting of two sorted arrays $A = [1, 1, 2, 2, 2, 2, 4, 5, 5, 8]$ and $B = [2, 2, 2, 2, 5, 10]$. In our example, we would bump A's pointer twice (for the 1s), then bump both pointers four times (for the 2s), bump A's pointer (for the 4), bump both pointers (for the 5), bump A's pointer (for the excess 5), bump A's pointer (for the 8), and then return "NO" (because we reached the end of A failing to find 10).

Correctness:

We need to argue that the algorithm always returns "YES" and "NO" correctly. The key idea is that the algorithm pairs up things in B with their counterparts in A. Since the arrays are sorted, you never have to look backwards, so you fail to find the counterpart if and only if it doesn't exist. We present a careful proof below: (many variations/arguments with similar ideas also exist)

There are two things to prove, namely: the algorithm terminates, and the algorithm gives the correct answer. The algorithm terminates because each iteration of the loop increments at least one of i and j, and we return if we reach the end of array A or B. So, it remains to prove that the algorithm outputs "NO" if and only if an element in B occurs less times in A than in B.

For the forwards direction, suppose the algorithm outputs "NO". Let the value of $B[i]$ at the time of returning be b, and assume b occurs in B exactly t times. We claim that b occurs in A less than t times. Suppose for contradiction this is false. Consider the first instance of the loop where $B[i] = b$, and let the values of i and j be m and k respectively. The arrays are sorted, hence for $i < m$, $B[i] \leq b - 1$. Also, j is only incremented when $A[j] \leq B[i]$. Thus, $A[k] \leq b - 1$. So, past index k, b occurs in A at least t times. We only return "NO" if j exceeds length(A) or $b < A[j]$. But, each iteration, i and j increase by at most 1, so neither situation can happen without first having $A[j] == B[i] == b$ exactly t times. Each time this occurs, i is incremented, thus, when we return "NO" we must have $i \geq m + t$. However, b appears in B exactly t times, with the first occurrence at index m, and B is sorted. Hence when we output "NO", the value of $B[i] > b$, which is a contradiction.

For the backwards direction, suppose $b \in B$ occurs k times in B but less times in A. Suppose b first appears in B at index m and assume for contradiction that our algorithm returns "YES". Note that $m + k \leq$ length(B)+1 and each iteration never increases i by more than 1. Thus, it is not possible to have i exceed length(B) before $i = m + k$. But the algorithm never returns "YES" before i exceeds length(B). Thus, there are two possibilities: either the algorithm terminates before $i = m + k$ and returns "NO" (a contradiction), or we reach $i = m + k$ via increments of i. However, i is only incremented when $B[i] == A[j]$ holds, and when this occurs, j is also incremented. So, for i to reach $m + k$, we must have found k occurrences of b in A, which contradicts our assumption.

Time Complexity:

The array is of size $\frac{n}{\sqrt{\log(n)}}$ and the number of digits is at most $\sqrt{\log(n)}$. So, Radix Sort with Counting Sort runs in $\Theta(\sqrt{\log(n)} \cdot \frac{n}{\sqrt{\log(n)}}) = \Theta(n)$.

In the second part of the algorithm, all operations in the loop are constant time. Also, each iteration, at least one of i and j is incremented, and if either reaches length(B) and length(A), the algorithm terminates. Thus, the loop costs at most length(A) + length(B). This is $\Theta(n)$ since length(A) is $\Theta(n)$ and length(B) is $O(n)$ because

$$\lim_{n\to\infty} \frac{n}{\frac{n}{\sqrt{\log(n)}}} = \lim_{n\to\infty} \frac{1}{\sqrt{\log(n)}} = 0$$

It follows that our algorithm is $\Theta(n)$.

Alternative Solutions:

Here are (without proof) some other approaches to the problem:

- Alt1: for each element in B, linearly search and mark as found in A. This will cost us

$$\Theta(\frac{n^2}{\sqrt{\log(n)}})$$

- Alt2: we could improve Alt1 by using binary search. This gives complexity

$$\Theta(\frac{n\log(n)}{\sqrt{\log(n)}}) = \Theta(n\sqrt{\log(n)})$$

- Alt3: we know that there are $\sqrt{\log(n)}$ digits so there are at most $10^{\sqrt{\log(n)}}$ numbers. We could show that $10^{\sqrt{\log(n)}} = O(n)$ and then form an array C which counts the number of occurrences of each element in B. Then, we can traverse A once and decrement C according to the values in A, and then do a single pass over C to verify that all array items are non-negative.